VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

FILTROVÁNÍ PAKETŮ V POČÍTAČOVÝCH SÍTÍCH

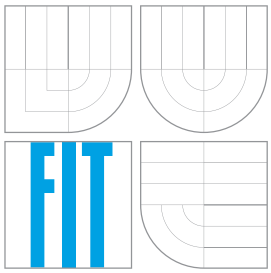
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JAN FARON

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

FILTROVÁNÍ PAKETŮ V POČÍTAČOVÝCH SÍTÍCH

PACKET FILTERING IN COMPUTER NETWORKS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JAN FARON

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MICHAL KAJAN

BRNO 2013

Abstrakt

Tato bakalářská práce se zabývá problematikou klasifikace paketů v počítačových sítích. V úvodu jsou popsány některé oblasti využívající klasifikaci paketů. Dále je uvedena potřebná teorie spolu s požadavky na klasifikační algoritmus. Jsou popsány čtyři vysokoúrovňové přístupy ke klasifikaci paketů. Ke každému přístupu jsou popsány principy několika algoritmů. Pro detailnější rozbor a implementaci je vybrán algoritmus EffiCuts. Tento klasifikační algoritmus je porovnán s jinými klasifikačními algoritmy z knihovny NetBench.

Abstract

This bachelor's thesis deals with packet classification in computer networks. In the introduction it describes some areas where packet classification is used. Then, necessary theoretical background is introduced, together with requirements for classification algorithm. It describes four high-level approaches to the packet classification. It describes principle of various algorithms for each high-level approach. Algorithm EffiCuts is chosen for detailed analysis and implementation. This packet classification algorithm is compared with other packet classification algorithms from NetBench library.

Klíčová slova

klasifikace paketů, efficuts, klasifikační algoritmus

Keywords

packet classification, efficuts, classification algorithm

Citace

Jan Faron: Filtrování paketů v počítačových sítích, bakalářská práce, Brno, FIT VUT v Brně, 2013

Filtrování paketů v počítačových sítích

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Michala Kajana

.....
Jan Faron
15. května 2013

Poděkování

Děkuji vedoucímu své bakalářské práce Ing. Michalovi Kajanovi, za ochotu a odbornou pomoc.

Obsah

Kapitola 1

Úvod

Tato bakalářská práce se zabývá problematikou klasifikace paketů v počítačových sítích především v oblasti internetu. Počítačové sítě využíváme v našich životech stále více. Společně s tím vznikají aplikace, které mají za úkol řídit data putující na internetu. Data se v počítačových sítích přenáší za pomocí paketů. Paket obsahuje kromě samotných dat i hlavičku paketu. V hlavičce paketu jsou důležité informace, které využívá mnoho aplikací a síťových zařízení, například směrovače. Úkolem směrovače je přeposlat vstupní paket na jedno ze svých výstupních rozhraní podle nejlepší shody cílové adresy se záznamem ve směrovací tabulce. Jednou z aplikací je systém zajišťující kvalitu služeb (QoS – Quality of Service), která má zajistit rychlý přenos například multimediálních dat. Další důležitá aplikace je firewall. Ten rozhoduje podle sady filtrovacích pravidel, jestli paket může projít dál nebo se zahodí. Podle filtrovacích pravidel je tedy možné zakázat nebo povolit určité typy komunikace.

Úkolem klasifikačního algoritmu je vyhledat nejlepší pravidlo, které odpovídá informacím z hlavičky paketu. Při dnešních rychlostech v počítačových sítích musí klasifikační algoritmus vyhledat nejlepší pravidlo z velké množiny pravidel ve velmi krátkém čase. Softwarová řešení klasifikačních algoritmů již dnes nedosahují potřebné rychlosti. Proto se dnes hledají řešení, která se dají implementovat v hardware.

V následující kapitole je představena problematika klasifikace. Ve třetí kapitole jsou popsány čtyři vysokoúrovňové přístupy ke klasifikaci a ke každému z nich jsou uvedeny principy několika algoritmů. Ve čtvrté kapitole je detailně popsán algoritmus EffiCuts, který vychází z algoritmu HyperCuts. Jsou zde vysvětleny všechny heuristiky, které tyto algoritmy používají. V páté kapitole je popsána implementace algoritmu EffiCuts. V šesté kapitole jsou uvedeny výsledky měření algoritmu EffiCuts. Dále je zde srovnání se svými předchůdci HyperCuts a HiCuts hlavně z pohledu paměťové náročnosti.

Kapitola 2

Problematika klasifikace

2.1 Klasifikační pravidla

Hlavička paketu obsahuje několik polí s hodnotami, které definují paket. Každé pole představuje jednu dimenzi. Klasifikace může probíhat v 1 až N dimenzích. Implementovaný algoritmus EffiCuts 4 klasifikuje pravidla ve standardních pěti dimenzích (zdrojová IPv4 adresa, cílová IPv4 adresa, zdrojový port, cílový port a protokol). Pravidlo určuje pro každou dimenzi určenou ke klasifikaci podmínku. Podmínka může být zapsána několika způsoby:

- **Prefix**
  Prefix tvoří několik prvních bitů, ve kterých se hodnota v dané dimenzi musí shodovat. Zbývající bity mohou být nastaveny libovolně. Typické použití tohoto zápisu je u IP adres, kde prefix představuje adresu sítě.
- **Rozsah**
  Je zadán intervalem hodnot. Hodnota v dané dimenzi musí patřit do zadaného intervalu. Tento způsob zápisu se často používá u zdrojového a cílového portu.
- **Hodnota**
  Hodnota v dané dimenzi se musí přesně shodovat se zadanou hodnotou. Typické použití je u protokolu.

Pokud paket projde všemi podmínkami, tak odpovídá danému pravidlu. Jedno pravidlo může vyhovovat velké množině paketů. Stejně tak jeden paket může odpovídat více pravidlům. Aby se mohlo určit, které pravidlo se má vybrat, musí pravidla obsahovat také prioritu. Pravidlo kromě priority a podmínek obsahuje i akci, která rozhodne, co se s paketem stane. Mezi typické akce patří zahození, propuštění nebo přesměrování paketu [1].

2.2 Klasifikační algoritmus

Klasifikace paketů představuje algoritmický problém, kde na vstupu jsou hodnoty z hlavičky paketu. Pro tyto hodnoty se vyhledá několik pravidel, které odpovídají daným hodnotám. Z těchto pravidel se následně vybere to s nejvyšší prioritou a vrátí se jako výsledek klasifikace.

2.2.1 Požadavky na klasifikační algoritmus

Rychlost vyhledání

Doba vyhledávání klasifikačního algoritmu nesmí být delší než doba mezi příchody jednotlivých paketů na lince. Nesmí tedy snižovat přenosovou rychlost sítě. Například pro linky o rychlosti 10 Gb/s musí být doba vyhledání nejlepšího pravidla kratší než 32 nanosekund [5].

Paměťová náročnost

Na různě velkou paměť potřebujeme různé typy pamětí. Obecně platí, že menší paměti jsou rychlejší. Rychlost přístupu do paměti u jednotlivých typů paměti značně ovlivňuje i rychlost vyhledávání, protože klasifikační algoritmy potřebují pro vyhledání pravidla několikrát přistoupit do paměti. Navíc velikost spotřebované paměti ovlivňuje celkovou cenu zařízení.

Kapitola 3

Přístupy ke klasifikaci paketů

Klasifikaci paketů můžeme rozdělit na čtyři vysokoúrovňové přístupy [13]. Každý algoritmus pro klasifikaci používá jeden, nebo kombinací z následujících přístupů.

1. **Kompletní prohledávání**
   Prozkoumá každé pravidlo z množiny pravidel.

2. **Rozhodovací strom**
   Z množiny pravidel se sestaví rozhodovací strom. Podle hodnot polí hlavičky paketu určených ke klasifikaci se prochází stromem do nalezení správného pravidla.

3. **Dekompozice**
   Místo prohledávání ve více dimenzích současně se každá dimenze prohledá samostatně. Pro nalezení správného pravidla se výsledky těchto samostatných prohledávání se zkombinují.

4. **Prohledávání n-tic**
   Rozdělí množinu filtrovacích pravidel do skupin n-tic podle počtu bitů (velikostí prefixů) z jednotlivých dimenzí určených ke klasifikaci, kde n je počet dimenzí. Prohledává se skupina n-tic přesným porovnáním, například pomocí hashovací funkce.

3.1 Kompletní prohledávání

3.1.1 Lineární prohledávání

Lineární prohledávání je nejvíce intuitivní přístup ke klasifikaci paketů. Všechna pravidla jsou uložena jen v jedné tabulce. Potom stačí každý paket postupně porovnávat se všemi pravidly v tabulce.

Výhodou tohoto algoritmu jsou minimální nároky na paměť, protože stačí uložit pouze samotná pravidla. Nevýhodou je lineární časová složitost, kvůli které se tento algoritmus stává pro větší sady pravidel velice nevhodný, protože každé pravidlo potřebuje jeden přístup do paměti. Rozdělíme-li ale pravidla na menší části, můžeme provést prohledávání pro každou část samostatně.

Lineární prohledávání je výhodné pro malé sady pravidel. V praxi se používá jako závěrečná fáze prohledávání v kombinaci s jiným přístupem, kde se počet pravidel může omezit podle vhodně zvolené hodnoty.

3.1.2 TCAM

Ternary Content Addressable Memory [6] je speciálně navrhnuté zařízení pro klasifikaci paketů. Jde o typ asociativní paměti, která dokáže paralelně porovnávat hodnoty z hlavičky paketu se všemi pravidly z celkové množiny pravidel. Vstupem je obecně klíč, který se porovnává s uloženými klíči pomocí komparátorů vyhodnocujících jejich shodu.

TCAM umožňuje kromě jedniček a nul ukládat stav zvaný *don't care*. Tento stav znamená, že na daném bitu může být hodnota **1** nebo **0** a na výsledek to nebude mít žádný vliv. Pro porovnávání je nejvhodnější převést pravidla na prefixy. Potom se jako klíč uloží hodnota prefixu doplněná stavem *don't care*.

TCAM mají i mnoho nevýhod. Mají velkou spotřebu energie, omezenou paměť, vysokou pořizovací cenu a jsou omezeny délkou vyhledávacího klíče. I přes své nevýhody jsou TCAM často používány. Tato technologie byla optimalizována příchodem Extended TCAM (E-TCAM), která výrazně snižuje spotřebu energie až o 90% tím, že aktivuje pouze určité oblasti. E-TCAM částečně využívá principu rozhodovacího stromu.

3.2 Rozhodovací strom

3.2.1 Rozhodovací strom pro jednu dimenzi

Klasifikace paketů pouze v jedné dimenzi má své využití zejména ve směrovačích, kde je potřeba příchozí paket předat na některé z jeho výstupních rozhraní podle cílové IP adresy. Směrovač musí vyhledat pro příchozí paket nejkonkrétnější pravidlo, ze své směrovací tabulky. Jelikož jsou pravidla převedena na prefixy, směrovač musí nalézt nejdelší shodný prefix ze směrovací tabulky a s cílovou IPv4 adresou příchozího paketu.

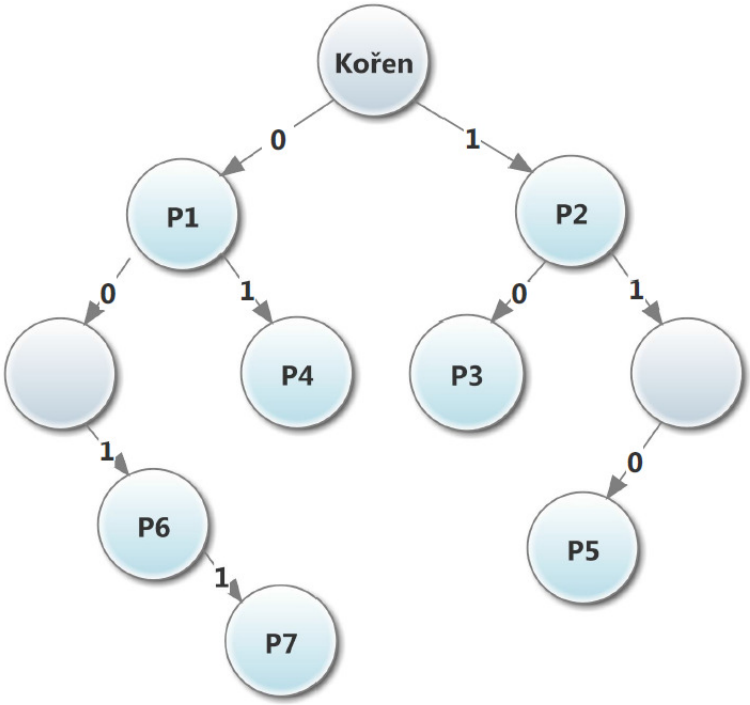
Prefixy IPv4 adresy jsou tvořeny bity o délce maximálně 32 bitů. To znamená, že maximální výška stromu může být 32. V praxi ale tento nejhorší případ často nenastává. Spíše nastává stav, kde prefixy jsou mnohem kratší (většinou délky 24 či kratší pro IPv4 adresy [5]). V tabulce 3.1 je ukázka převodů IPv4 adres na prefixy.

Číslo pravidla	**Prefix**	**IPv4 adresa**
P1	0*	0.0.0.0/1
P2	1*	128.0.0.0/1
P3	10*	128.0.0.0/2
P4	01*	64.0.0.0/2
P5	110*	192.0.0.0/3
P6	001*	32.0.0.0/3
P7	0011*	48.00.0.0/4

Tabulka 3.1: Tabulka prefixů

Jedním z nejvíce používaných algoritmů zabývajícím se problematikou LPM (Longest Prefix Match) je binární stromová struktura *trie*. Název *trie* pochází z anglického slova *retrieval* (vyhledání). Ve stromové struktuře *trie* se prefixy z tabulky pravidel ukládají přímo do uzlů stromu. Struktura stromu *trie* je ilustrována na obrazku 3.1.

Stromová struktura se začíná stavět od kořene stromu, kde se po ohodnocených hranách bity **1** nebo **0** tvoří další uzly podle prefixů z tabulky pravidel. Každý vnitřní uzel stromu má tedy jednoho nebo dva potomky. Pokud uzel odpovídá danému prefixu, tak se do uzlu



Obrázek 3.1: Stromová struktura *trie*

uloží číslo pravidla daného prefixu. Platí tedy, že kratší prefix může být součástí delšího prefixu.

Ve stromu se vyhledává podle binární formy klasifikovaného pole (IP adresy) z hlavičky paketu. Postupuje se od nejvíce významného bitu (MSB - Most Significant Bit) k méně významným bitům. Při vyhledávání se postupně od kořene stromu po ohodnocených hranách podle bitů z klasifikovaného pole z hlavičky paketu prochází uzly, které mohou obsahovat platný prefix (číslo pravidla). Vyhledávání končí, narazí-li se na list stromu. Poslední platný prefix, na který se při procházení od kořene stromu narazí, se vždy zapamatuje a je následně vrácen jako výsledek vyhledávání.

Časová složitost vyhledávání v binární stromové struktuře *trie* je lineární $\mathcal{O}(W)$, kde W je délka nejdelšího prefixu v tabulce. Paměťová složitost je $\mathcal{O}(N \cdot W)$, kde N je počet všech prefixů v tabulce.

3.2.2 Rozhodovací stromy pro více dimenzí

Rozhodovací strom umožňuje klasifikaci ve více dimenzích. Rozhodovací strom se sestrojí tak, že listy stromu obsahují jedno nebo více pravidel a vnitřní uzly obsahují informace, potřebné k rozhodnutí, jakým dalším uzlem se bude při prohledávání pokračovat. Pole z hlavičky paketu určená ke klasifikaci se použijí jako klíč při průchodu stromem. Z klíče se použije jeden nebo více bitů pro určení dalšího uzlu.

U méně konkrétních pravidel často dochází k takzvané replikaci pravidla. To znamená, že pravidlo bude uloženo ve více listech stromu. Časová náročnost je lineární s počtem bitů, které obsahuje klíč. Algoritmy postavené na bázi rozhodovacího stromu, přinášejí

řadu optimalizací nad tímto základním a ne příliš efektivním principem. Jedním z nich je algoritmus EffiCuts, který je blíže představen v kapitole 4.

3.2.3 Síť stromů (Grid of tries)

Sít stromů je rozšíření binární stromové struktury *trie* [11] používané pro klasifikaci v jedné dimenzi, na použití ve více (nejčastěji ve dvou) dimenzích. Klasifikace ve dvou dimenzích má své uplatnění například na páteřních směrovačích, které obsahují velké množství pravidel typu cílová IP adresa- zdrojová IP adresa pro směrování IP adres, VPN sítí či multicastového vysílání [5].

Síť stromů vytváří pro první dimenzi jeden strom *trie*. Uzly z prvního stromu *trie* obsahující prefixy z první dimenze a ukazují na další stromy *trie* pro vyhledávání v druhé dimenzi. Jako příklad dosadíme za první dimenzi cílovou IP adresu a za druhou dimenzi zdrojovou IP adresu.

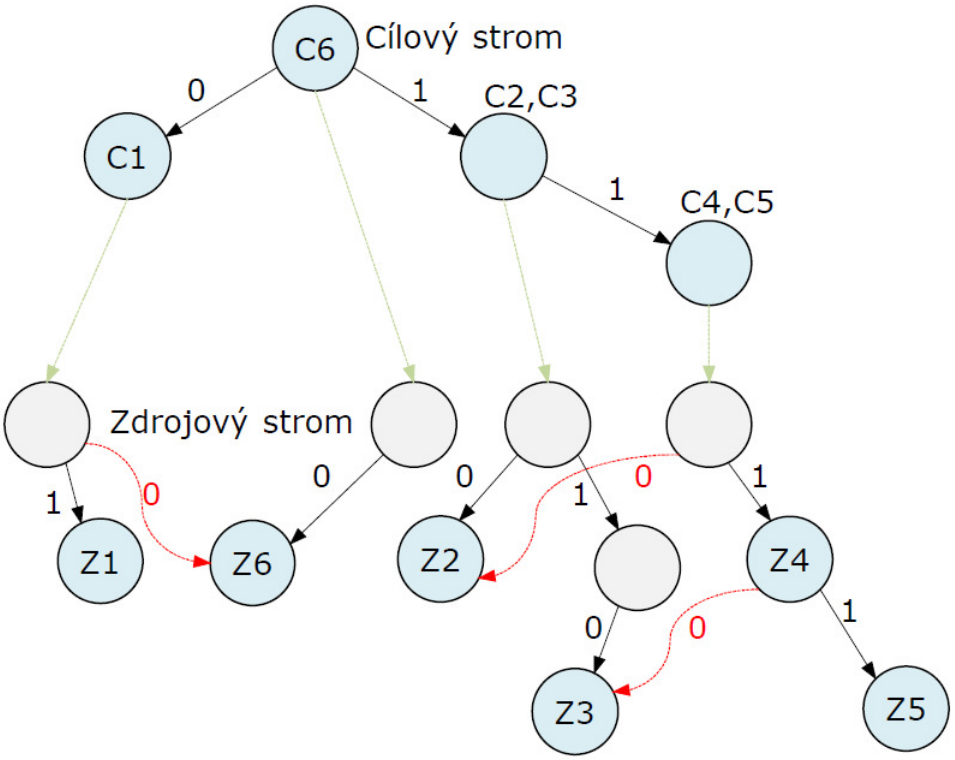
Základní princip při vyhledávání v síti stromů začíná vyhledáváním podle první dimenze, kde získáme například nejdelší prefix pro cílovou IP adresu. Poté probíhá vyhledávání ve stromu pro druhou dimenzi například zdrojové IP adresy. V aktuálním stromě pro druhou dimenzi mohou být kopírovány další stromy z druhé dimenze, které jsou odkazovány z předchozích prefixů první dimenze.

Vyhledávání v takové datové struktuře je sice rychlé, ale paměťové nároky jsou kvůli velké replikaci pravidel příliš velké. Na způsob jak zamezit velkým paměťovým nárokům přišla varianta stromu *trie* se zpětným vyhledáváním, kde je každé pravidlo uloženo pouze jednou. Tato metoda se liší ve způsobu vyhledávání ve stromu. Vyhledávat se začíná opět od první dimenze. Když se postupně prohledává první dimenze, prochází se i uzly s kratšími prefixy než se vyhledá nejdelší shodný prefix první dimenze. Pokud se nenalezne hledaný prefix ve stromu druhé dimenze, algoritmus začne prohledávat i další stromy druhé dimenze, na které ukazují právě uzly s kratším prefixem z první dimenze. Proces vyhledávání bude ale časově náročnější než v prvním případě. Ve stromu z obrázku 3.2 má-li vstupní paket v cílové i zdrojové IP adrese prefix 0*, tak se nejdříve vyhledá nejdelší shodný prefix C1 v cílovém stromu, ale ve zdrojovém stromu se nenajde žádný shodný prefix. Algoritmus se vrací na kratší prefix v cílovém stromu C6 a ve zdrojovém stromu už najde prefix Z6.

S další optimalizací přichází varianta se zpětným vyhledáváním s ukazateli. Tento algoritmus vylepšuje proces vyhledávání předchozí varianty se zpětným vyhledáváním. Při nenalezení prefixu ve stromu z druhé dimenze se algoritmus nevrací, ale použije předvypočítané ukazalete na určitý uzel dalšího stromu z druhé dimenze. Tímto způsobem se zlepší časová složitost z $\mathcal{O}\left(W^2\right)$ na $\mathcal{O}\left(2 \cdot W\right)$, kde W je maximální počet bitů hledané dimenze, při zachování lineární paměťové složitosti. Na obrázku 3.2 se použijí předvypočítané ukazatele.

Pravidlo	Cílový prefix	Zdrojový prefix
P1	0*	1*
P2	1*	0*
P3	1*	10*
P4	11*	1*
P5	11*	11*
P6	*	0*

Tabulka 3.2: Tabulka prefixů



Obrázek 3.2: Grid of tries se zpetnými ukazateli.

Pokud jsou obě vyhledávání úspěšná, tak došlo k nalezení pravidla s nejdelším shodným prefixem v cílové i zdrojové adrese. V opačném případě hledaný prefix nevyhovuje žádnému pravidlu a paket je zpravidla zahozen.

3.3 Dekompozice

Vzhledem k tomu, že vyhledávání v jedné dimenzi je velice efektivní, vznikla myšlenka na rozdělení vyhledávání ve více dimenzích na vyhledávání v jednotlivých dimenzích, které se zpracovávají nezávisle na sobě. Výsledek každé části se musí určitým způsobem zpracovat, abychom ze všech výsledků získali pouze jedno nejspecifičtější pravidlo.

Hlavní výhodou je možnost optimalizovat vyhledávání pro každou dimenzi samostatně a ve všech dimenzích vyhledávat paralelně. Hlavní problém tohoto přístupu je určit výsledné pravidlo z těch pravidel, která byla výsledná pro jednotlivé dimenze. Vyhledáváme-li ve všech dimenzích například podle nejdelšího shodného prefixu, tak tyto dílčí výsledky nejsou jednotné. Proto se často jako dílčí výsledek nevrací jen jedno pravidlo, ale vrací se množina pravidel. Dílčí výsledky se zkombinují a vybere se pravidlo, které není jen nejlepší v jedné dimenzi, ale vyhovuje i pro ostatní dimenze. Tento přístup hlavně díky možnosti paralelně zpracovat jednotlivé kroky poskytuje nízkou časovou náročnost.

3.3.1 Vyhledávání pomocí bitového vektoru

Základem této metody [4, 5] je použití bitového vektoru, který znázorňuje výskyt pravidel v určitém prefixu. Pro každý prefix je předem předvypočítán bitový vektor, který obsahuje právě tolik bitů kolik je pravidel. V bitovém vektoru je nastavena hodnota určitého bitu na 1 pokud pravidlo pokrývá daný prefix, jinak je bit nastaven na hodnotu 0. Filtrovací pravidla musí být seřazena podle priority, aby se z bitového vektoru dalo určit, které pravidlo se má použít.

Vyhledávání má dvě části. V první části se v každé dimenzi vyhledá nejdelší shodný prefix. Dále se z tabulky pro danou dimenzi zjistí podle vyhledaného prefixu předvypočítaný bitový vektor. V druhé části se nad vyhledanými bitovými vektory provede bitová operace AND, neboli průnik bitových vektorů. Výsledný bitový vektor znázorňuje výskyt pravidel, která klasifikovaný paket pokrývají. Jelikož jsou pravidla seřazena podle priority, tak výsledné pravidlo určuje první nejvíce významný bit z výsledného vektoru nastavený na 1.

Pravidlo	Dimenze 1	Dimenze 2
P1	00*	11*
P2	0*	00*
P3	1*	0*

Dimenze 1	Bitový vektor
0*	010
00*	110
1*	001

Dimenze 2	Bitový vektor
0*	001
00*	011
11*	100

Tabulka 3.3: Bitové vektory pro jednotlivé prefixy ze dvou dimenzí.

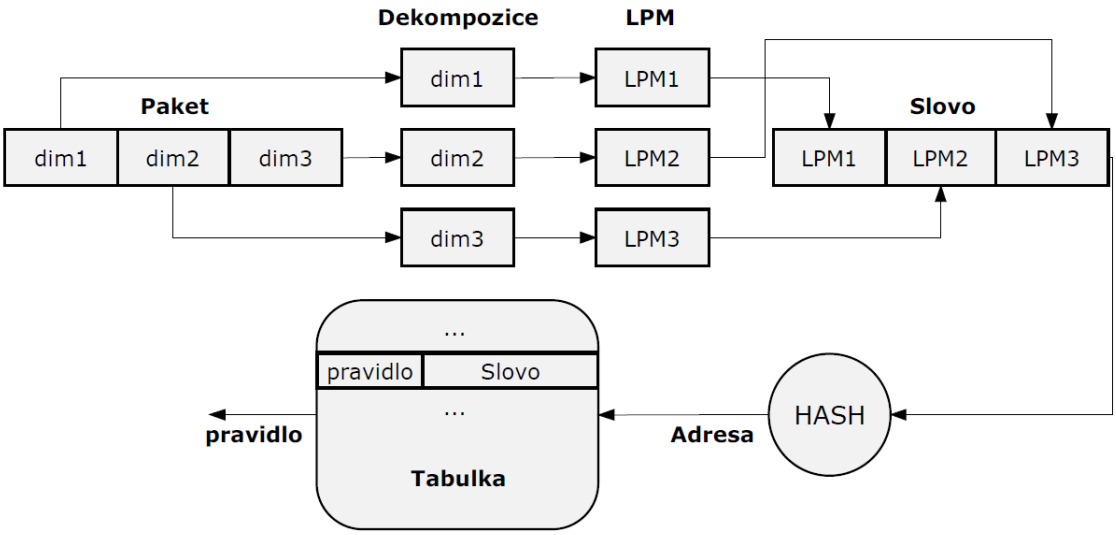
Časová složitost tohoto algoritmu je $\mathcal{O}(N)$, kde N je velikost bitového vektoru. Časová složitost je nezávislá na počtu dimenzí. To je způsobeno tím, že se vyhledává v jednotlivých dimenzích paralelně. Tato metoda neumožňuje efektivní přidávání a ubírání pravidel, protože se musí celá datová struktura znovu přepočítat.

3.3.2 Kartézský součin

Na základě pozorování se zjistilo, že počet různých prefixů v jednotlivých dimenzích je mnohem menší než celkový počet prefixů v jednotlivých dimenzích [13]. Například množina pravidel o 100 pravidlech obsahuje pouze 22 různých zdrojových adres, 17 různých cílových adres a 11 různých zdrojových portů. Na základě tohoto zjištění vznikla myšlenka pro každou možnou kombinaci unikátních prefixů z jednotlivých dimenzí předpočítat nejlepší pravidlo. Předpočítaný nejlepší výsledek z kombinace nejlepších výsledků pro jednotlivé dimenze uložíme do tabulky na místo, které nám vrátí hashovací funkce, když ji dáme na vstup slovo, které vznikne spojením nejlepších výsledků z jednotlivých dimenzí. V tabulce je uložen výsledek pro každou kombinaci z možných výsledků z jednotlivých dimenzí. To má za následek velké paměťové nároky. V nejhorším případě má exponenciální paměťovou složitost $\mathcal{O}\left(N^k\right)$, kde N je počet pravidel a k je počet dimenzí. Jednou z možností jak ušetřit paměť je vytvářet tabulku postupně až v průběhu vyhledávání.

Vyhledávání pomocí kartézského součinu probíhá následovně. Pole z hlavičky příchozího paketu určená ke klasifikaci se rozdělí a zpracují samostatně. V každé dimenzi se vypočítá

nejlepší výsledek pro danou dimenzi například metodou vyhledání nejdelšího shodného prefixu. Výsledky z jednotlivých dimenzí jsou spojeny do jednoho slova, které slouží jako vstup do hashovací funkce. Výstupem hashovací funkce je adresa do tabulky obsahující nejlepší pravidlo pro vstupní paket. Obrázek 3.3 znázorňuje tento algoritmus.



Obrázek 3.3: Karterský součin.

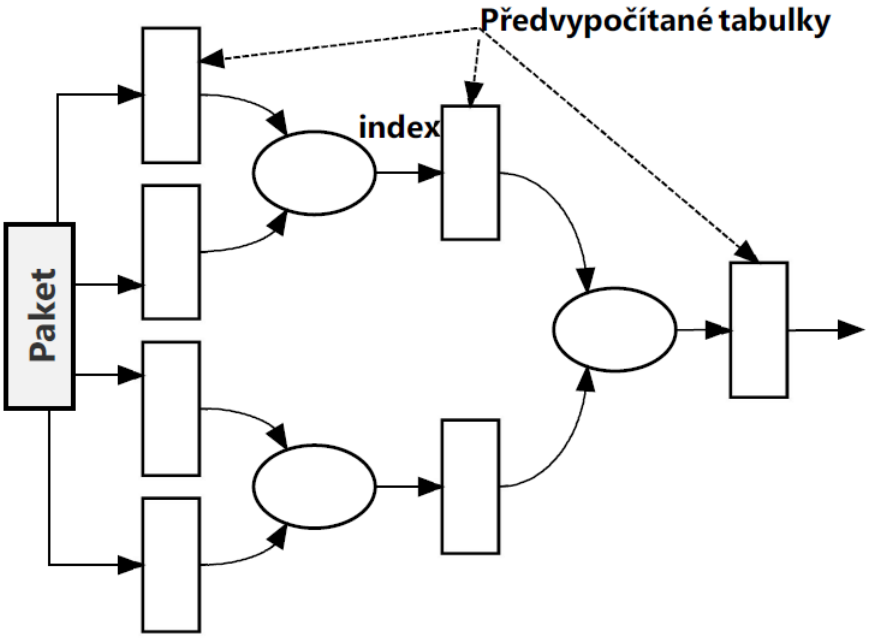
3.3.3 Rekurzivní klasifikace toků RFC

Tento algoritmus [7, 5, 9, 2] vychází z metody kartézského součinu, kde pro všechny možné kombinace výsledků získaných z jednotlivých dimenzí se předpočítá nejlepší výsledné pravidlo pro danou kombinaci. Všechny možné kombinace jsou uloženy v jedné velké tabulce.

RFC Recursive Flow Clasification místo jednoho kartézského součinu nad všemi dimenzemi, který má za následek jednu velkou tabulku, provádí dílčí kartézské součiny (například pro dvě dimenze) nad třídami ekvivalence. RFC rozšiřuje metodu kartézského součinu o bitový vektor, který reprezentuje třídu ekvivalence. Třída ekvivalence je množina kombinací vektorů (prefixů) ze dvou částí (dimenzí), které odpovídají stejné množině pravidel. To znamená, že mají stejný bitový vektor. Velikost tabulek je potom výrazně menší než u kartézského součinu ze všech dimenzí. Vzniklá třída ekvivalence může být dále poslána na vstup dalšího dílčího kartézského součinu jak je vidět na obrázku 3.4. Takto se tvoří hierarchická struktura, kde na konci je tabulka obsahující množinu pravidel. Tento algoritmus má velice dobré výsledky co se týče časové náročnosti, ale za cenu vysokých nároků na paměť.

3.4 Prohledávání n-tic

Tento přístup ke klasifikaci [10, 9] rozdělí množinu filtrovacích pravidel do skupin n-tic podle počtu bitů (velikostí prefixů) z jednotlivých dimenzí určených ke klasifikaci, kde n je počet dimenzí. Například u klasifikace v pěti dimenzích je n-tice [4, 3, 8, 16, 0] (zdrojová adresa,



Obrázek 3.4: RFC Recursive Flow Clasification.

cílová adresa, protocol, zdrojový port, cílový port). Aby pravidlo patřilo do takové n-tice musí mít zdrojová adresa délku prefixu 4, cílová adresa délku prefixu 3, protokol musí být jeden konkrétní, zdrojový port musí být jeden konkrétní a cílový port může být jakýkoliv.

Z pozorování se zjistilo, že počet různých n-tic je mnohem menší než počet filtrovacích pravidel. Navíc pravidla v jedné n-tici se v prostoru nepřekrývají. Právě díky unikátnosti pravidel se prohledává skupina n-tic přesným porovnáváním v jednotlivých n-ticích, například pomocí hashovací funkce. Vstupem do hashovací funkce je slovo, které vznikne spojením bitů z jednotlivých dimenzí dané n-tice, například 4 bity ze zdrojové adresy + 3 bity z cílové adresy a tak dále. Výstupem je adresa do tabulky filtrovacích pravidel. Vrátí-li se více pravidel, vybere se to s největší prioritou.

3.4.1 Prořezávané prohledávání n-tic

Klasický způsob musí prohledávat všechny existující n-tice. Počet těchto n-tic může být velmi velký. Prořezávané prohledávání n-tic se zaměřuje na zredukování počtu n-tic, které je nutné prohledat pro získání správného pravidla. Tento algoritmus používá vyhledávání nejdelšího shodného prefixu LPM v několika dimenzích. V praxi typicky zdrojová a cílová adresa. Tím se vybere jen určité množství kandidátních n-tic, které je nutné prohledat.

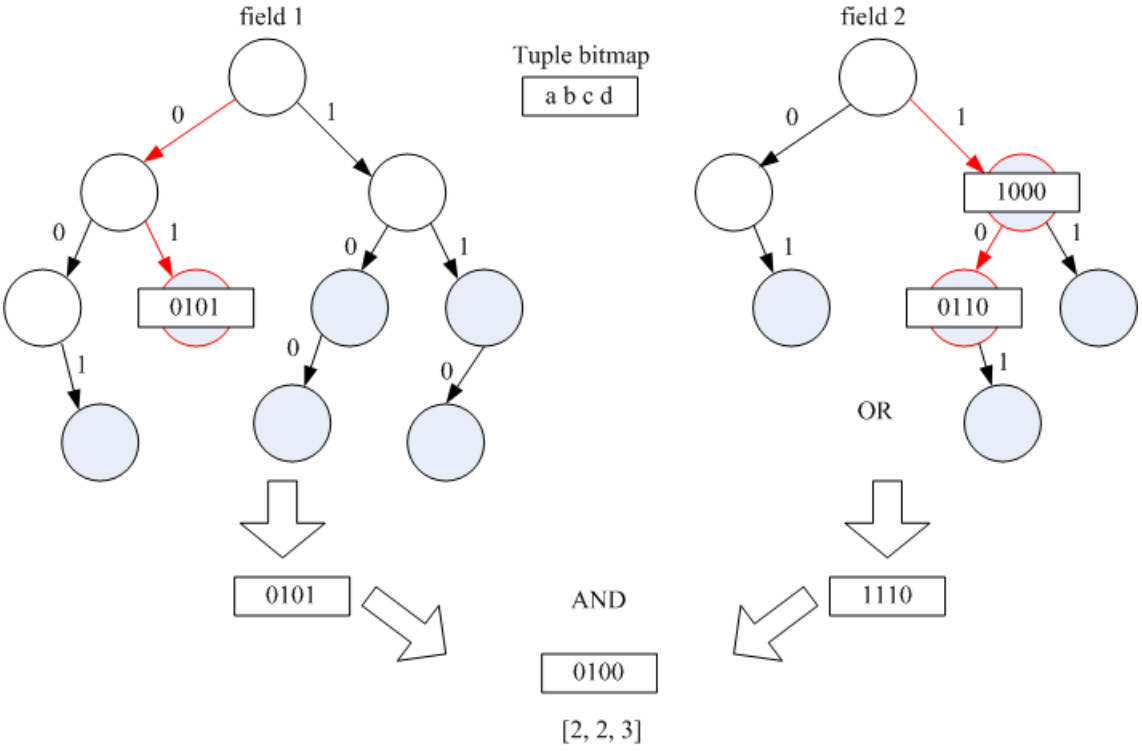
Na příkladu převzatém z [9] se z tabulky 3.4 vyhledává nejdelší shodný prefix v dimenzi 1 a 2 jak je ukázáno na obrázku 3.5. V uzlech stromu mohou být uloženy bitové vektory reprezentující, které n-tice se mají prohledávat. Při průchodu stromem se zapamatují všechny nalezené bitové vektory a provede se nad nimi bitové OR. Bity nastavené na 1 znamenají, že daná n-tice se musí prohledat. Například bitový vektor 1110 znamená, že se musí prohledat

n-tice a, b, c. Nad výslednými bitovými vektory obou stromu se provede bitové AND. Tímto průnikem se zjistí n-tice platné pro oba stromy. Nakonec se prohledají všechny n-tice, které reprezentuje výsledný bitový vektor.

Číslo pravidla	dimenze 1	dimenze 2	dimenze 3	n-tice
P1	001*	1*	11*	[3, 1, 2]
P2	01*	10*	010*	[2, 2, 3]
P3	100*	10*	011*	[3, 2, 3]
P4	11*	01*	011*	[2, 2, 3]
P5	110*	11*	101*	[3, 2, 3]
P6	10*	01*	111*	[2, 2, 3]
P7	11*	101*	110*	[2, 3, 3]

ID n-tice	n-tice	Čísla pravidel
a	[3, 1, 2]	1
b	[2, 2, 3]	2, 4, 6
c	[3, 2, 3]	3, 5
d	[2, 3, 3]	7

Tabulka 3.4: Převod na n-tice.



Obrázek 3.5: Stromové prohledávání n-tic [9].

Kapitola 4

EffiCuts

4.1 Obecný princip algoritmu

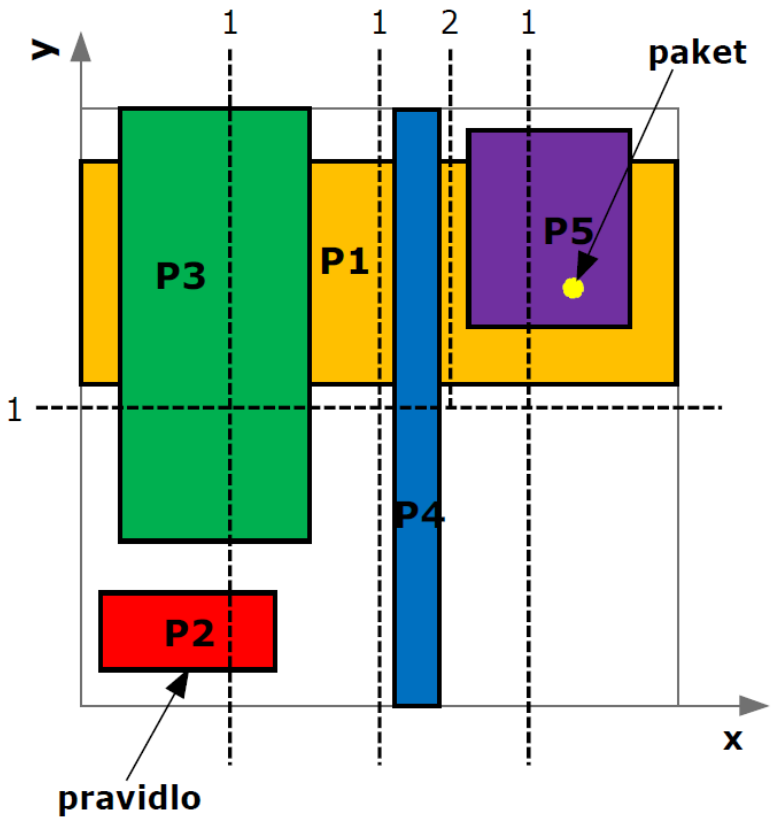
EffiCuts [3] je klasifikační algoritmus, který vychází z algoritmu HyperCuts [12] a jeho předchůdce HiCuts [8]. Na klasifikaci pohlíží jako na geometrický problém, kde každé pole z hlavičky paketu určené pro klasifikaci představuje jednu dimenzi. Každý paket tedy představuje bod a každé pravidlo představuje obecně vícerozměrný objekt (hyperkvádr) ve vícedimenzionálním diskrétním pravidlovém prostoru. Pravidlový prostor pro dvě dimenze je ilustrován na obrázku 4.1.

Všechny tři algoritmy spočívají v konstrukci rozhodovacího stromu. Rozhodovací strom se vytvoří postupným dělením pravidlového prostoru do více stejně velkých (*equi-sized*) podprostorů. Dělení pravidlového prostoru začíná od kořenového uzlu, který pokrývá celý prostor. Při každém dělení vznikají další uzly reprezentující podprostory, které se dále mohou dělit nezávisle na sobě. Čím jsou uzly ve stromu hlouběji, tak pokrývají čím dál tím menší části pravidlového prostoru a tím pádem obsahují čím dál tím méně pravidel. Často nastává situace, kde pravidlo pokrývá více podprostorů. V takovém případě nastává takzvaná replikace pravidla. To znamená, že se pravidlo zkopíruje do všech uzlů, které pravidlo pokrývá. Jakmile uzel obsahuje méně pravidel než je hodnota prahu (*bucket-size*), stává se listem. V listech jsou uložena samotná filtrovací pravidla nebo ukazatele na ně (záleží na konkrétním algoritmu).

Konstrukce rozhodovacího stromu pro pravidlový prostor z obrázku 4.1 je ilustrovaný na obrázku 4.2. Na obrázku je pravidlový prostor rozdělen v prvním kroku na 4 části podle **osy x** a na 2 části podle **osy y**. Tím vznikne 8 nových uzlů. V druhém kroku se dále rozdělí jeden z nových uzlů na 2 části podle **osy x**, je-li nastavena hodnota *bucket-size*, která reprezentuje maximální počet pravidel v listu stromu na 2.

Existují tedy dva typy uzlů. Vnitřní uzel, který slouží k navigaci průchodu stromem při klasifikaci a vnější uzel (list), který obsahuje množinu pravidel. Následná klasifikace spočívá v průchodu stromem, kde se podle hodnot polí v hlavičce paketu a podle toho, jak byl uzel rozdělen, rozhodne, kterým z jeho potomků se bude pokračovat. Průchod končí nalezením listu stromu a následně se z množiny pravidel, které list obsahuje, lineárně vyhledá pravidlo s nejvyšší prioritou pokrývající paket.

Po klasifikačním algoritmu chceme nízkou paměťovou náročnost a vysokou rychlost vyhledání pravidla. V rozhodovacím stromu představuje paměťovou náročnost počet uzlů a počet pravidel pro klasifikaci. Zatím co rychlost vyhledávání představuje výška stromu spolu s počtem pravidel v listu, protože čas vyhledání se určuje podle počtu přístupů do paměti. Chceme-li zlepšit jednu z těchto vlastností, tak to obvykle negativně ovlivní tu druhou.



Obrázek 4.1: Pravidlový prostor pro 2 dimenze.

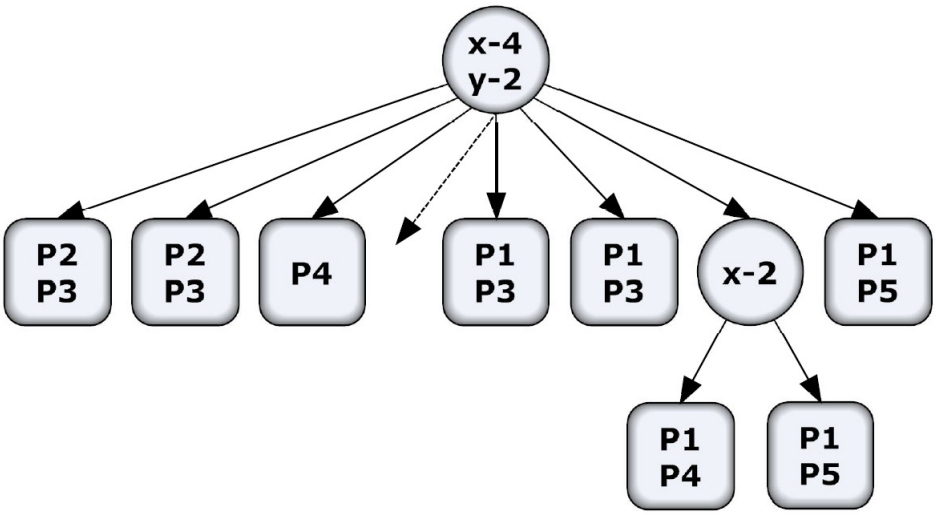
Cílem tohoto přístupu je rozdělení pravidel do listů stromu co nejrovnoměrněji, při co nejmenší výšce stromu a s co nejmenším počtem uzlů. Toho je obecně velmi těžké dosáhnout, a proto se používají různé heuristiky, které pomáhají sestavit rozhodovací strom co nejoptimálněji. Jelikož EffiCuts je postaven na algoritmu HyperCuts 4.2 a využívá z něho téměř všechny heuristiky, je nutné si tento algoritmus blíže popsat.

4.2 HyperCuts

HyperCuts tvoří základ pro implementovaný algoritmus EfiiCuts. Tento algoritmus je založený na principu dělení pravidlového prostoru na stejně velké podprostory a vytváří se tak rozhodovací strom. Tento princip je popsaný v předchozí kapitole 4.1.

V každém uzlu u HyperCuts může probíhat dělení prostoru do menších podprostorů podle více než jedné dimenze v každém kroku. Musí se tedy rozhodnout podle které dimenze a na kolik částí se bude prostor dělit. Pravidlový prostor se kvůli snadnější implementaci dělí na mocninu čísla 2 podprostorů.

HyperCuts má dva konfigurovatelné parametry. *Space-factor*, který se používá k určení maximálního počtu potomků a *bucket-size*, který určuje maximální počet pravidel, aby mohl být uzel označený jako list. Větší velikost *bucket-size* může snížit hloubku stromu,



Obrázek 4.2: Stromová struktura, která vznikne po dělení pravidlového prostoru z obrázku 4.1.

ale prodlužuje lineární vyhledávání v listu. Optimální nastavení těchto parametrů závisí na množině pravidel, pro kterou je strom sestavěn. Zde jsou popsány heuristiky HyperCuts [9].

4.2.1 Výběr dimenzí, podle kterých se bude pravidlový prostor dělit

Hloubku stromu z pravidla nejvíce ovlivňuje potomek s nejvíce pravidly. Proto se tato heuristika snaží rozdělit pravidla co nejrovnoměrněji mezi potomky. Toho se docílí výběrem těch dimenzí, ve kterých jsou pravidla prostorově nejvíce rozmístěná. Jinými slovy k dělení se vyberou ty dimenze, které mají větší počet unikátních rozsahů než průměrný počet unikátních rozsahů ze všech dimenzí. Například pro klasifikaci v pěti dimenzích, kde unikátní rozsahy v jednotlivých dimenzích jsou znázorněny v tabulce 4.1 se vyberou dimenze zdrojová IPv4 adresa a Cílová IPv4 adresa na rozdělení pravidlového prostoru, protože průměrný počet unikátních rozsahů je 10.

Zdrojová adresa	**Zdrojový port**	**Cílová adresa**	**Cílový port**	**Protokol**
14	8	**22**	4	2

Tabulka 4.1: Příklad unikátních rozsahů v jednotlivých dimenzích.

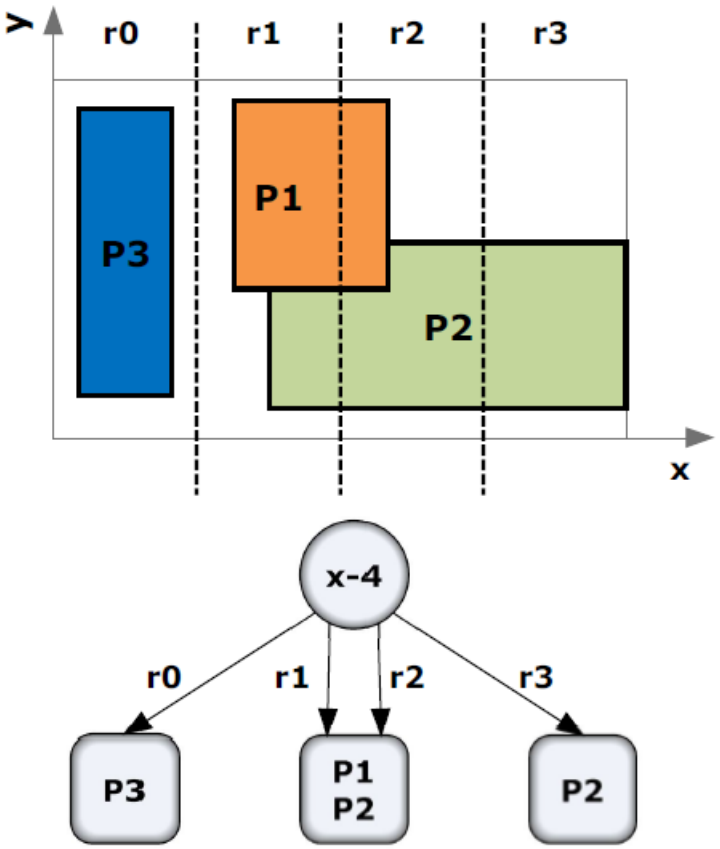
4.2.2 Určení na kolik částí se každá vybraná dimenze rozdělí

Je důležité si uvědomit, že čím více řezů provedeme, tím více vznikne uzlů a je větší šance, že dojde k replikaci pravidel. To má za následek větší paměťovou náročnost. Na druhou stranu se zmenšuje výška stromu. Je tedy potřeba zjistit kolik může mít uzel maximálně potomků, aby časová náročnost vyhledávání a paměťová náročnost byla optimální. Maximální počet potomků uzlu NC se vypočítá jako $NC = \textit{space-factor} \cdot \sqrt{N}$, kde *space-factor*

je konfigurovatelný parametr a N je počet filtrovacích pravidel, které uzel obsahuje.

4.2.3 Node merging (sjednocení uzlů)

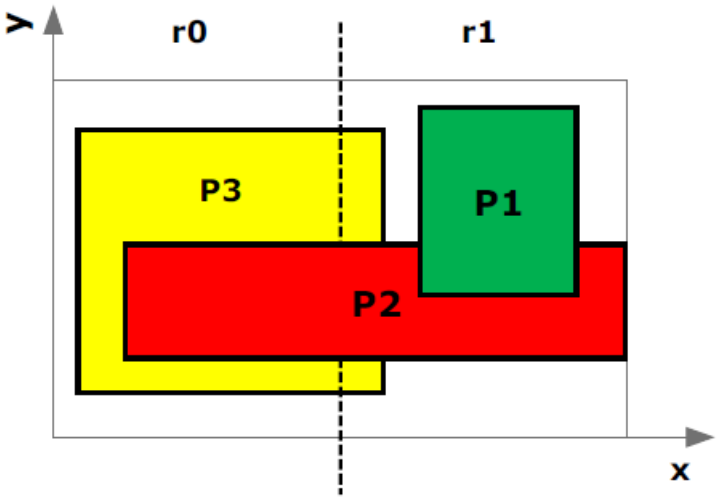
Při dělení uzlu mohou někteří jeho potomci obsahovat stejnou množinu pravidel se svými sourozenci. Místo toho aby se vytvářel uzel pro každého z nich, vytvoří se jen jeden a ostatní sourozenci se stejnou množinou pravidel si k němu uchovávají jen ukazatel. Na obrázku 4.3 je ukázán příklad sjednocení uzlů, kde uzly z regionu 1 a 2 obsahují stejnou množinu pravidel. Vytvoří se pouze jeden uzel, ale ukazatele na něj budou dva.



Obrázek 4.3: Příklad sjednocení uzlů.

4.2.4 Filter overlap (překrytí celého pravidla pravidlem s vyšší prioritou)

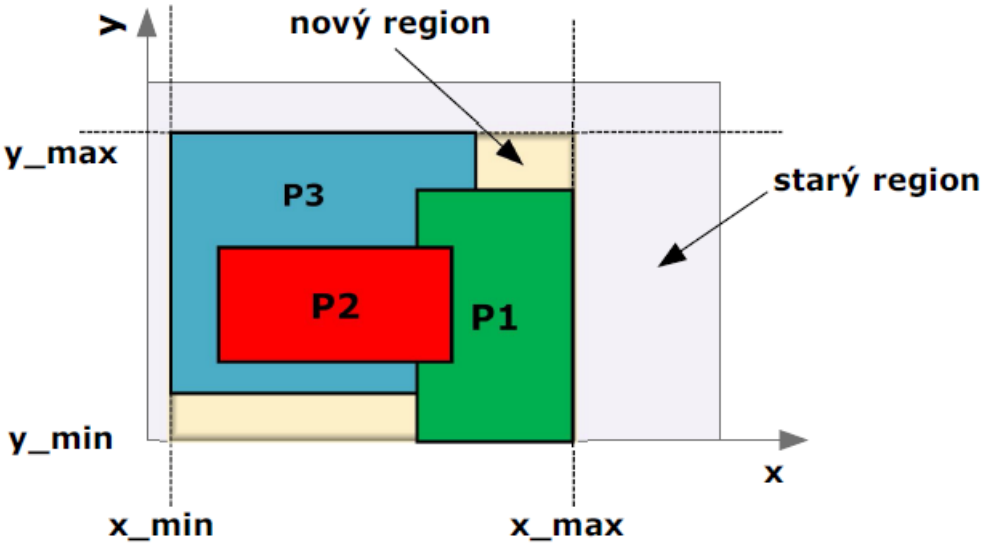
Při dělení může nastat situace, kde v jednom podprostoru bude existovat pravidlo s vyšší prioritou, které bude plně překrývat pravidlo s nižší prioritou. V takovém případě se z uzlu odebere pravidlo s nižší prioritou, které by se nikdy nemohlo uplatnit. Tím se zmenší paměťová náročnost a někdy se může snížit i výška stromu. Na obrázku 4.4 pravidlo P3 pokrývá celé pravidlo P2. Pokud pravidlo P3 má větší prioritu než pravidlo P2, tak se pravidlo P2 odebere z uzlu pokrývající region 0.



Obrázek 4.4: Příklad překrytí pravidel po dělení uzlu. Pravidlo P3 plně pokrývá pravidlo P2.

4.2.5 Region compaction (zmenšení prostoru)

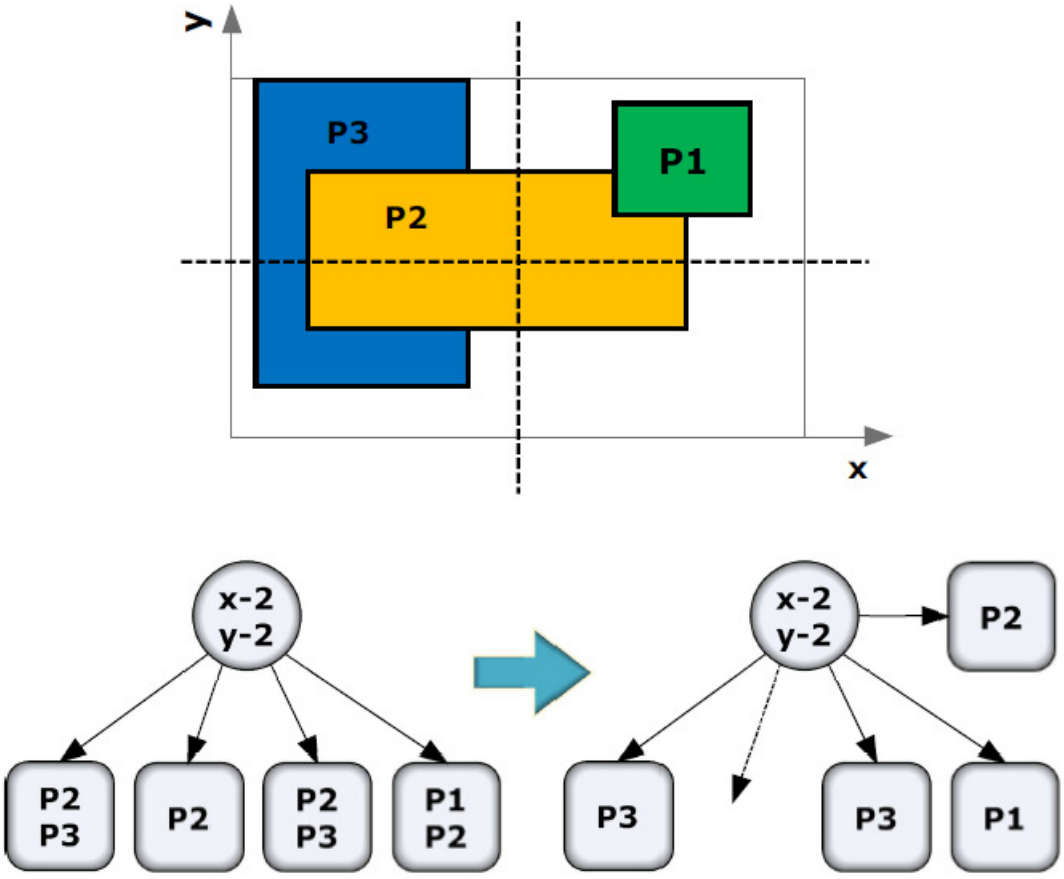
Pravidlový prostor, který uzel reprezentuje, může být větší než minimální prostor, který pokrývá všechna pravidla v daném uzlu. Obrázek 4.5 ilustruje zmenšení pravidlového prostoru. Zmenšíme-li prostor na minimální, tak následující dělení pravidlového prostoru bude efektivnější. Nevýhodou je, že si každý uzel musí uchovávat hranice prostoru, který reprezentuje.



Obrázek 4.5: Příklad zmenšení pravidlového prostoru.

4.2.6 Moving-up

Existuje-li případ, kde všichni potomci rodičovského uzlu sdílí některá pravidla. Všechna tato pravidla se přesunou do rodičovského uzlu, jak je znázorněno na obrázku 4.6, kde společné pravidlo P2 je přesunuto do rodičovského uzlu. Tato heuristika je aplikována, až v době kdy je strom postaven. Od zdola nahoru rekurzivně posouvá společná pravidla pro všechny sourozence do rodičovského uzlu. Výsledkem je sice snížení replikace pravidel, ale při vyhledávání ve stromu přidá jeden přístup do paměti ke každému vnitřnímu uzlu pro přístup ke společným pravidlům.

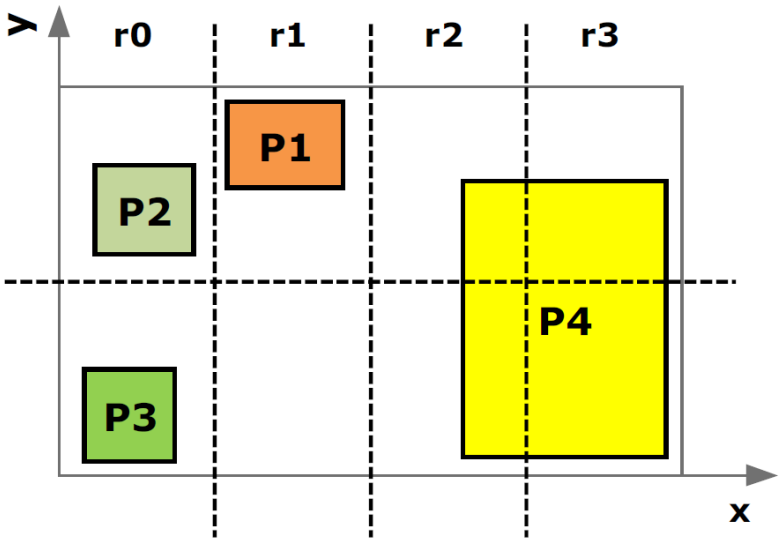


Obrázek 4.6: Příklad heuristiky Moving-up. Společne pravidlo je posunuto do rodičovského uzlu.

4.3 Heuristiky EffiCuts

EffiCuts přináší další čtyři optimalizace nad algoritmem HyperCuts, které znatelně snižují paměťovou náročnost, ale i rychlost vyhledávání. Při použití HyperCuts hlavně pro velké sady pravidel (100 000) spotřebuje i přes všechny optimalizace značně velkou paměť. První příčinou je překrývání velkého množství pravidel v prostoru. Když chceme rozdělit pravidla pokrývající malou část prostoru (malá pravidla), tak musíme provést více řezů, aby se

prostor rozdělil na malé oblasti, jak můžeme videt na obrázku 4.8. To ovšem replikuje pravidla pokrývající velkou část prostoru (velká pravidla). To má za následek neefektivní velký strom. Druhou příčinou je různá hustota pravidel v prostoru. Potřebujeme-li rozdělit malá pravidla blízko u sebe a navíc jsou v malé části prostoru, vznikne mnoho neefektivních uzlů, které obsahují jen pár pravidel. Obrázek 4.7 znázorňuje různou hustotu pravidel v pravidlovém prostoru. Na tyto příčiny velké paměťové náročnosti se zaměřují heuristiky EffiCuts. *Separable Trees* společně se *Selective Tree Merging* řeší první problém a *Equi-dense cuts* řeší druhý problém.



Obrázek 4.7: Různá hustota pravidel v prostoru. Při dělení prostoru vzniká mnoho neefektivních uzlů

Nejdříve budou představeny principy těchto heuristik. Následně se každá z nich prozkoumá do větších detailů. Cílem první z nich *Separable Trees* je oddělit překrývající se malá a velká pravidla v prostoru. Pravidla se rozdělí do kategorií a podkategorií podle toho, jestli jsou malá, nebo velká pro každou z dimenzí. Pro každou podkategorii se vytvoří rozhodovací strom zvlášť. Tohle řešení razantně sníží replikaci pravidel za cenu toho, že se při klasifikaci musí procházet všechny stromy a tím se zvýší časová náročnost vyhledání pravidla.

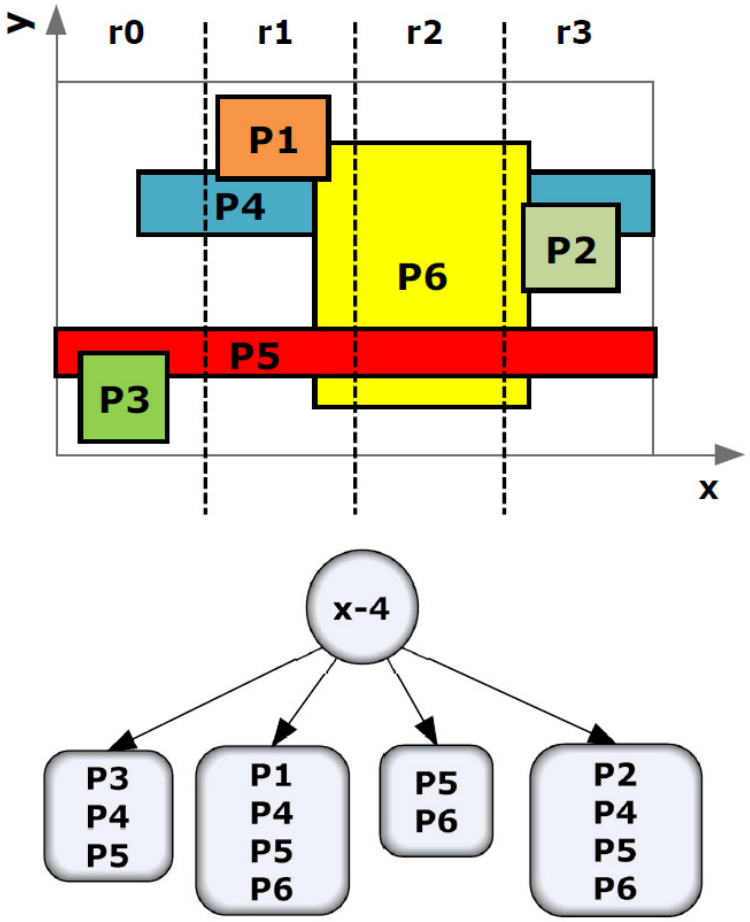
Aby snížení propustností nebylo tak významné, zajišťuje druhá heuristika *Selective Tree Merging*. Ta se snaží zpátky sjednotit některé stromy, u kterých by docházelo k replikaci pravidel jen minimálně. Sjednocení dvou stromů může probíhat jen v případě, kde je odlišnost velkých a malých pravidel pouze v jedné dimenzi.

Třetí heuristika *Equi-dense cuts* vytvoří efektivnější uzly sjednocením několika neefektivních za sebou jdoucích uzlů. Nevýhodou je o trochu složitější vyhledávání.

Čtvrtá heuristika *Node Co-location* přidává část (hlavičku) uzlu potomka k ukazateli na něj. Tím zredukuje počet přístupů do paměti ze dvou na jeden pro každý uzel, který používá *Equi-dense cuts*.

4.3.1 Separable Trees

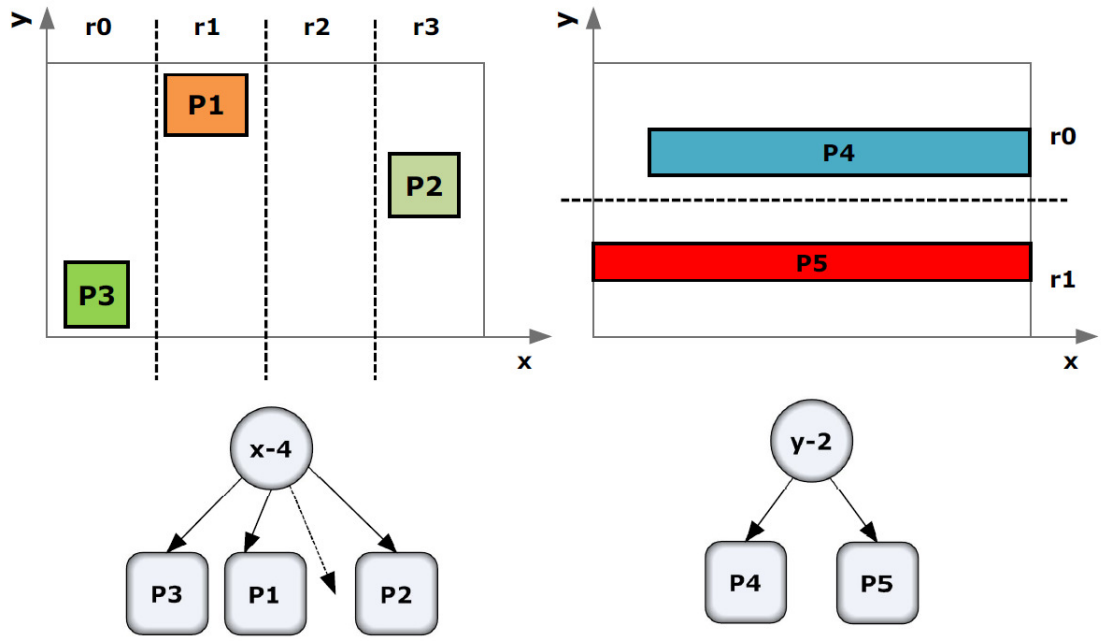
Cílem této heuristiky je předejít replikaci pravidel při překrývání malých a velkých pravidel v prostoru. Na obrázku 4.8 je ilustrováno překrývání pravidel v prostoru. Pokud bychom potřebovali rozdělit malá pravidla P1, P2 a P3, tak by došlo ke značné replikaci velkých pravidel P4, P5 a P6.



Obrázek 4.8: Překrývání malých a velkých pravidel v prostoru vede k replikaci pravidel.

Separable Trees rozděluje pravidla do kategorií a podkategorií podle toho, jestli jsou malá, nebo velká pro každou z dimenzí. Pro každou podkategorii se vytvoří rozhodovací strom zvlášť. To je ukázáno na obrázku 4.9, kde se množina všech pravidel rozdělí do podmnožin {P1,P2,P3}, {P4,P5} a {P6}. Následně se pro každou podmnožinu postaví strom zvlášť. Tím se výrazně sníží replikace pravidel a tedy i celková paměťová náročnost. Na druhou stranu je nutné projít všechny stromy pro klasifikaci jednoho paketu a tím se zvyšuje časová náročnost vyhledávání.

Pravidla se roztřídí do kategorií podle toho, v kolika dimenzích jsou velká. Kategorie se dále dělí na podkategorie podle toho, v jakých dimenzích bylo pravidlo velké. Klasifikujeme-li podle standardních pěti dimenzí (zdrojová ip adresa, zdrojový port, protokol, cílová ip adresa, cílový port), tak třídíme pravidla do následujících kategorií.



Obrázek 4.9: Pravidla se rozdělí do kategorií a pro každou z nich se postavý strom zvlášť.

- **Kategorie 1**, je-li pravidlo velké ve čtyřech dimenzích (pět podkategorií)
- **Kategorie 2**, je-li pravidlo velké ve třech dimenzích (deset podkategorií)
- **Kategorie 3**, je-li pravidlo velké ve dvou dimenzích (deset podkategorií)
- **Kategorie 4**, je-li pravidlo velké v jedné, nebo žádné dimenzi (nedělí se)

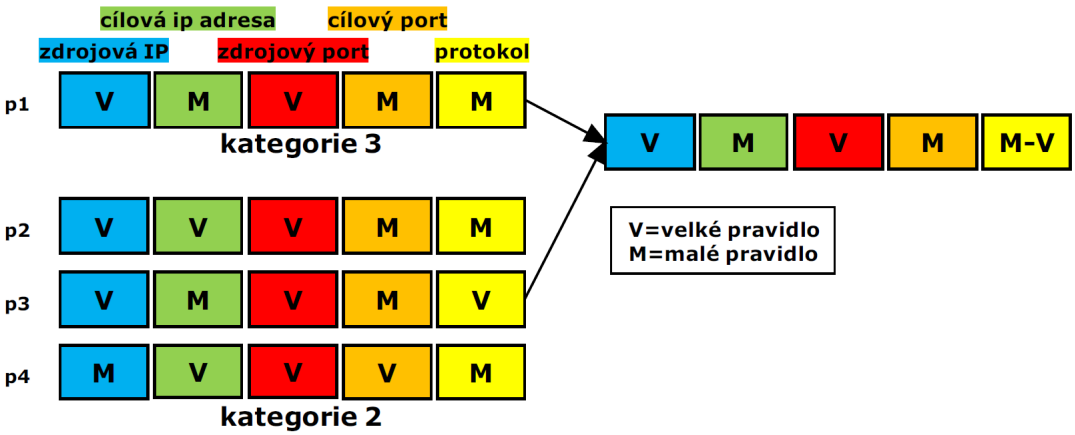
Kategorie 4 se nedělí na podkategorie, protože obsahuje převážně malá pravidla. Celkem existuje 26 podkategorií. Mnoho z nich je obvykle prázdných. Pro rozdělení pravidel v každé dimenzi na malé a velké slouží hranice *largeness-fraction*. Ta v podstatě udává, kolik procent z celkové velikosti dimenze musí pokrývat, aby se mohlo pravidlo označit v této dimenzi za velké. U IP adres musí zabírat alespoň 5% rozsahu, aby bylo označeno za velké. U ostatních dimenzí má *largeness-fraction* hodnotu 0,5 tedy 50% rozsahu dimenze.

4.3.2 Selective Tree Merging

Separable Trees rozdělilo pravidla do několika podkategorií. Kdyby byl pro každou podkategorii sestavěn strom, tak by každý paket musel projít všechny stromy. To znamená více přístupů do paměti a tím pádem i větší časová náročnost.

Selective Tree Merging se snaží o dobrý kompromis mezi zvýšením propustnosti a snížením paměťových nároků tím, že sjednotí dvě podkategorie, které se liší jen v jedné dimenzi. Z toho vyplívá, že strom z kategorie i se může sjednotit pouze s vhodným stromem z kategorie $i + 1$, nebo z kategorie $i - 1$. To neplatí pro kategorii 4, kde jsou smíchány velké a malé pravidla v různých dimenzích, proto už kategorii 4 s nikým nesjednocujeme. Stejně tak už jednou sjednocené stromy dále nesjednocujeme, protože by byla porušena podmínka

odlišnosti malých a velkých pravidel pouze v jedné dimenzi. Na obrázku 4.10 je předveden pokus o sjednocení podkategorie p1 z kategorie 3 s některou z podkategorií p2, p3 a p4 z kategorie 2. Můžeme zde vidět, že podmínka odlišnosti pouze v jedné dimenzi platí pouze na podkategorie p2 a p3. Podkategorie p4 se liší už ve dvou dimenzích.



Obrázek 4.10: Příklad sjednocení kategorií

Z pozorování se zjistilo [3] , že sjednocení stromů, které mají méně pravidel obvykle končí s menším počtem replikujících se pravidel. Dále se zjistilo, že stromy obsahují více pravidel, jak jdeme od kategorie 1 do kategorie 4. Sjednocovat stromy tedy začínáme od kategorie 2. Potom kategorie i se snaží sjednotit s kategorií 1. Dále se pokračuje sjednocováním kategorie 3 s kategorií 2.

Najde-li se více jak jeden vhodný strom ke sjednocení, vybere se ten, který se liší v nejmenší dimenzi [1], protože u menších dimenzí je menší šance na replikaci pravidel. Na obrázku 4.10 se pro sjednocení vybere z podkategorií p2 a p3 právě p3, protože se liší s podkategorií p1 v menší dimenzi.

4.3.3 Equi-dense Cuts

Prostor se vždy dělí na mocninu čísla 2 podprostorů. To zjednodušuje vyhledání potomka. Potom stačí podle počtů řezů v uzlu vypočítat index do pole potomků obsahující mocninu čísla 2 položek.

Equi-dense cuts se zaměřuje na odstranění neefektivních uzlů, které vzniknou dělením různě hustého pravidlového prostoru. Potřebujeme-li rozdělit malá pravidla blízko u sebe a navíc jsou v malé části prostoru, vznikne mnoho neefektivních uzlů, které obsahují jen pár pravidel. Přitom mnoho z nich obsahuje stejnou, nebo částečně stejnou množinu pravidel. Heuristiky z HyperCuts sice sjednotí uzly obsahující stejnou množinu pravidel (*Node merging*), ale už se nevypořádá s částečně stejnou množinou pravidel. Navíc si každý uzel musí udržovat ukazatele na potomka v každé položce v poli potomků, které zabírá 30%-50% celkové paměti [3].

[1] IPv4 adresa o velikosti 32 bitů je větší než port o velikosti 16 bitů nebo protokol o velikosti 8 bitů

Equi-dense cuts se snaží sjednocovat za sebou jdoucí neefektivní uzly. Tím dosáhne daleko rovnoměrnějšího rozdělení pravidel mezi potomky, kterých je výrazně méně. Tato heuristika také sjednotí redundantní ukazatele v poli potomků, které se zmenší. To má za následek nejednotnost čísla indexu přicházejícího pro určení potomka s číslem položky ve zmenšeném poli potomků. Uzly používající *Equi-dense cuts* si musí navíc ukládat počet svých potomků a pole počátečních indexů sjednocených uzlů. Při vyhledávání potomka se z pole počátečních indexů se vybere index hodnoty, který je menší nebo rovno nejbližší hodnotě indexu vytvořeného paketem. Příklad *Equi-dense cuts* je na obrázku 4.11

Pro implementaci v hardwaru je nutné vědět, kolik bude maximálně potřebovat komparátorů. Zvolí se tedy hodnota maximálního množství potomků *max-cuts*. Jestliže se tato hodnota překročí, tak se algoritmus vrací ke klasickému dělení (*equi-sized*).

Existují dvě metody pro sjednocování uzlů. První metoda sjednotí za sebou jdoucí listy stromu, pokud výsledný počet pravidel nepřesáhne hranici *bucket-size*. Tato metoda snižuje počet uzlů a snižuje replikaci pravidel. Neovlivňuje však výšku stromu.

Druhá metoda sjednotí za sebou jdoucí vnitřní uzly, pokud 1. výsledný počet pravidel je menší než v obou uzlech dohromady. Tím se zajistí sjednocení stejné a částečně stejné množiny pravidel. 2. výsledný počet pravidel musí být menší než maximální počet pravidel, které má jeden uzel mezi všemi sourozenci. Tím se sníží šance na zvýšení výšky stromu.

Následkem přepočítávání indexů se lehce zvýší časová náročnost vyhledávání, ale v porovnání s ušetřenou pamětí se tato heuristika vyplatí.

4.3.4 Node Co-location

Při průchodu stromem potřebujeme nejméně dva přístupy do paměti. První přístup do paměti potřebujeme pro hlavičku uzlu, která obsahuje počet řezů v každé dimenzi a u uzlů používajících equi-dense cuts navíc pole počátečních indexů. Z těchto informací se vypočítá index do pole potomků. Právě přístup do pole potomků vyžaduje druhý přístup do paměti.

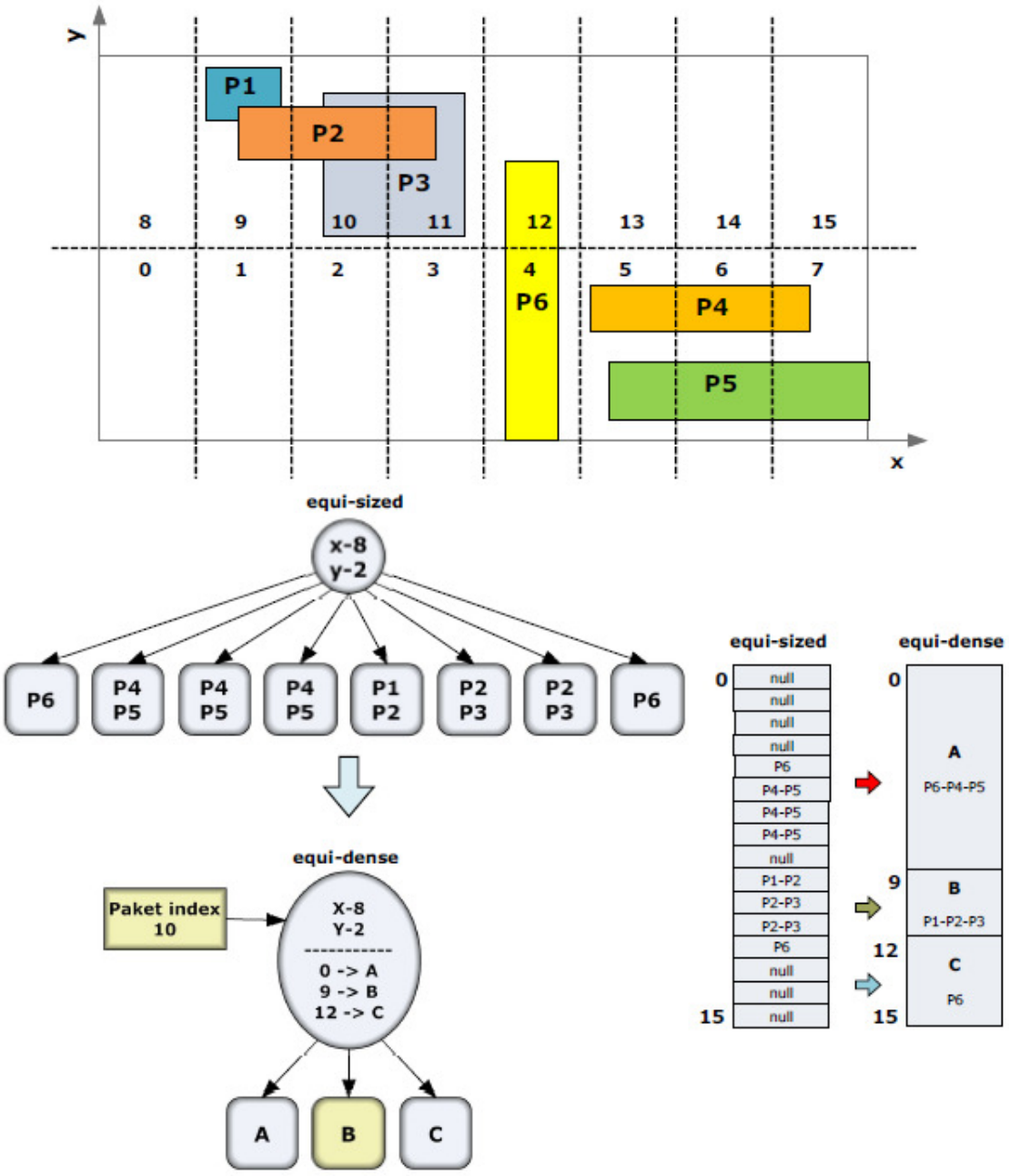
Cílem Node Co-location je zredukovat počet přístupů do paměti v každém uzlu ze dvou na jeden tím, že do každé položky v poli potomků přidá k ukazateli na potomka i jeho hlavičku. Stačí tedy přistupovat pouze do pole potomků, jak je ukázáno na obrázku 4.12. Hlavička uzlu ovšem zabírá další paměť. EffiCuts nepoužívá heuristiku moving-up, protože by znamenala další přístup do paměti.

Uzly používající equi-dense cuts mají počet potomků omezen hranicí max-cuts (obvykle 8). Zato uzly používající klasické dělení (equi-sized cuts) mají mnohem více potomků. Navíc díky heuristice z HyperCuts Node merging jsou některé ukazatelé stejné. Přidání hlaviček ke každému ukazateli by mělo za následek výrazný nárůst paměťových nároků. Z tohoto důvodu se Node Co-location používá pouze u uzlů používajících equi-dense cuts.

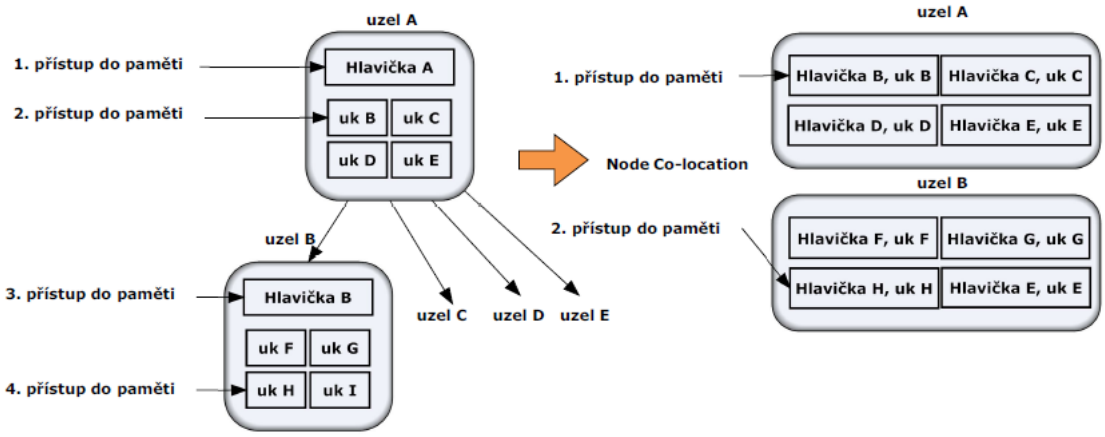
4.3.5 Zhodnocení heuristik

Heuristiky, se kterými přišel EffiCuts téměř odstranily replikaci pravidel, kvůli kterým si HyperCuts musel uložit všechna pravidla do jedné tabulky. V listech stromu byly uloženy pouze ukazatele na pravidla. EffiCuts si může dovolit ukládat do listů stromů celé pravidla (13 bytů) místo ukazatelů (4 bytů). Přitom paměťová náročnost bude více či méně stejná.

Uložíme-li do paměti za sebou celá pravidla, bude pro nalezení správného pravidla stačit pouze jeden přístup do paměti místo několika u HyperCuts. To je další velká výhoda zvláště proto, že se musí procházet více stromů a tím pádem i více listů.



Obrázek 4.11: Equi-dense cuts.



Obrázek 4.12: Node Co-location. Hlavička paketu potomka se přidá k ukazateli na něj. Tím se zredukuje potřebný počet přístupů do paměti.

Kapitola 5

Implementace

Algoritmus EffiCuts byl implementován v jazyku python za použití frameworku netbench [14]. Jako podklad pro implementaci byl použit algoritmus HyperCuts z tohoto frameworku, který byl upraven a rozšířen o optimalizace EffiCuts. Algoritmus podporuje standardní klasifikaci pro pět dimenzí (zdrojová IPv4 adresa, zdrojový port, protokol, cílová IPv4 adresa, cílový port).

5.1 Rozdělení pravidel

Pro pohodlnější a efektivnější práci s množinou pravidel je potřeba si vstupní množinu pravidel upravit. To znamená především odebrat pravidla, která jsou redundantní, anebo jsou pokryta pravidlem s vyšší prioritou.

5.1.1 Separable trees

Cílem *Separable trees* je rozdělit množinu pravidel na podmnožiny tak, aby se malá a velká pravidla v jednotlivých dimenzích překrývala co nejméně. Tím se výrazně zabrání replikaci pravidel, což vede ke snížení paměťové náročnosti.

Pro rozhodnutí, zda je pravidlo malé, nebo velké se stanový hodnoty *largeness-fraction* pro každou z dimenzí. Zdrojová a cílová IPv4 adresa má 32 bitů. Celkový rozsah je $2^{32} = 4294967296$. Aby bylo pravidlo označeno za velké, musí pokrývat alespoň 5% celkového rozsahu. $4294967296 \cdot 0,05 = 214748364,8$. Zdrojový a cílový port má 16 bitů. Tyto dimenze mají celkový rozsah $2^{16} = 65536$. Hodnota *largeness-fraction* je pro tyto dimenze 50% celkového rozsahu, tedy $65536 \cdot 0,5 = 32768$. Protože hodnota protokolu nenabývá žádný rozsah, je tato dimenze označena za velkou, pouze pokud obsahuje všechny protokoly (protocol any).

Nyní se všechna pravidla roztřídí do kategorií a podkategorií. Roztřídit pravidla do kategorií je velice snadné. Stačí si jen spočítat, v kolika dimenzích bylo pravidlo velké. Určit podkategorii znamená zjistit, v jakých dimenzích bylo pravidlo velké. Proto je přiřazena každé dimenzi váha (jednička v násobcích deseti). Největší váhu mají největší dimenze.

- Zdrojová IPv4 adresa: 10000
- Cílová IPv4 adresa: 1000
- Zdrojový port: 100

- Cílový port: 10
- Protokol: 1

Součet těchto vah určí výsledný vektor podkategorie. Například je-li pravidlo velké v dimenzích zdrojová IPv4 adresa a zdrojový port, vektor podkategorie bude 10100. Tato podkategorie patří do kategorie 3.

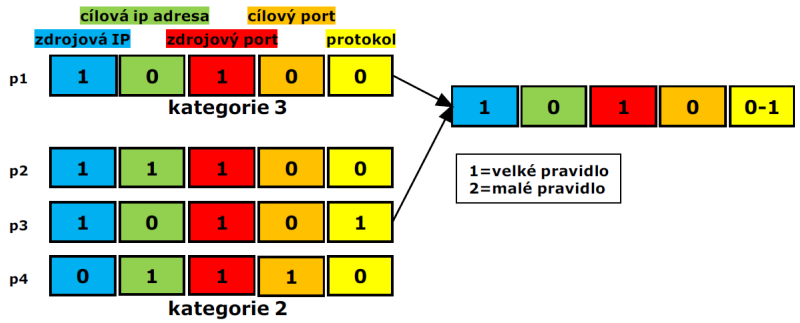
5.1.2 Selective Tree Merging

Nyní jsou všechna pravidla roztříděna do kategorií a podkategorií. Úkolem *Selective Tree Merging* je sjednotit vhodné podkategorie a tím snížit počet výsledných stromů. Z teoretické části víme, že sjednocení podkategorií s menším počtem pravidel obvykle vede k menšímu počtu replikací pravidel, a že počet pravidel v kategoriích roste od kategorie 1 ke kategorii 4. Pro úplnost jsem doplnil kategorii 0, kde je pravidlo ve všech dimenzích velké.

Algoritmus začíná výběrem podkategorie s nejmenším počtem pravidel z kategorie 1. Pro tuto podkategorii se z kategorie 0 hledá nejvhodnější kandidát pro sjednocení. Nejvhodnější kandidát pro sjednocení musí splňovat dvě podmínky:

1. Kategorie se musí lišit pouze v jedné dimenzi.
2. Pokud je kandidátů více, tak se vybere ten, který se liší v nejmenší dimenzi.

Tyto informace se dají zjistit pouhým rozdílem vektorů podkategorie. Aby byl rozdíl pouze v jedné dimenzi, tak rozdíl vektorů musí být jedno z množiny čísel $\{10000, 1000, 100, 10, 1\}$. Jelikož menší dimenzi reprezentuje menší váha, tak se jako nejvhodnější podkategorie vybere ta, která má nejmenší rozdíl čísel podkategorií z těchto čísel. Na obrázku 5.1 je ukázka sjednocení podkategorie p1 z kategorie 3 s podkategorií p3 z kategorie 2. Podkategorie p3 se vybrala, protože má nejmenší rozdíl vektorů s podkategorií p1 $10101 - 10100 = 1$.



Obrázek 5.1: Sjednocení podkategorií

S tímto principem se pokračuje mezi jednotlivými kategoriemi. Obecně podkategorie z kategorie i se snaží sjednotit s podkategorií z kategorie $i-1$. Vyjimkou je kategorie 4, která se už s nikým nesjednocuje, protože jsou v ní smíchána malá a velká pravidla v různých dimenzích a došlo by tedy k porušení podmínky rozdílu malých a velkých pravidel pouze v jedné dimenzi. Nyní se pro každou podmnožinu pravidel sestaví rozhodovací strom.

5.2 Stavba rozhodovacího stromu

Rozhodovací strom se staví rekurzivním voláním funkce `create_node()`. Jako první se vytvoří kořen stromu, pokrývající celý pravidlový prostor. Uzel se stává listem, pokud má stejně, nebo méně pravidel jako hranice *bucket-size*. Každý vnitřní uzel se nejdříve zkouší rozdělit podle *equi-dense cuts*.

Funkce začíná výběrem vhodných dimenzí, podle kterých se bude pravidlový prostor dělit. Dále se zjistí počet řezů v jednotlivých vybraných dimenzích. Pravidlový prostor se rozdělí na podprostory podle rovnoměrného dělení (*equi-sized cuts*). Rozdělení dimenze na rovnoměrné části probíhá tak, že se zvětší maska daná dimenze o tolik bitů, kolik je řezů v dané dimenzi a z každé nové kombinace bitů vznikne nový podprostor. Například prostor je dělen dvěma řezy. Maska dané dimenze se zvětší o 2 bity na 11. Potom vzniknou čtyři nové podprostory 00, 01, 10, 11. Pro každý podprostor se určí pravidla, která ho pokrývají. Každý podprostor reprezentuje třída *Child* uchovávající hranice prostoru a pravidla pro daný podprostor. Všichni potomci jsou uložený v poli potomků za sebou tak, jak byl prostor dělen.

Nyní zkoušíme sjednotit vždy první dva potomky z pole potomků. Jestli oba potomci budou mít méně pravidel než je hodnota *bucket-size* (budou listy stromu) zkoušíme je sjednotit. Podmínka pro sjednocení listů je velice jednoduchá. Výsledná množina pravidel, která vznikne sjednocením množin pravidel z obou listů, musí obsahovat méně pravidel než je hodnota *bucket-size*. Jinak řečeno ze dvou listů musí vzniknout opět list.

Podobně se zkouší sjednotit potomci, pokud budou oba vnitřní uzly. Podmínka pro sjednocení má dvě části:

1. Výsledná množina pravidel, která vznikne sjednocením množin pravidel z obou potomků, musí být menší než součet počtu pravidel z obou potomků.

2. Výsledná množina pravidel musí být menší než největší množina pravidel některého ze sourozenců.

Jsou-li tyto podmínky splněny, tak se musí sjednotit i ohraničení pravidlového prostoru obou potomků. Nenastane-li ani jedno sjednocení, přidá se index druhého potomka do tabulky indexů a pokračuje se sjednocováním druhého potomka se třetím a tak dále.

Po dokončení sjednocování, se pro každého zbylého potomka vytvoří nový uzel a ukazatel společně s hlavičkou nového uzlu se přidají do pole potomků aktuálního uzlu. *Equi-dense cuts* může skončit neúspěšně, pokud byl překročen limit maximálního množství potomků *max-cuts*. V tom případě se prostor rozdělí pomocí rovnoměrného dělení (*equi-sized cuts*).

Equi-sized cuts se na rozdíl od *equi-dense cuts* snaží zmenšit hranice svého pravidlového prostoru na minimum. Tím pádem bude rozdělení prostoru efektivnější. Na druhou stranu si uzel musí ukládat hranice svého prostoru. Při dělení prostoru se po provedení řezu rekurzivně vytvoří další uzel. Do pole potomků se vkládá pouze ukazatele na ně bez jejich hlaviček.

5.3 Vyhledávání v rozhodovacím stromu

Před vlastním vyhledáváním se musí z hlavičky příchozího paketu získat hodnoty z jednotlivých polí určených ke klasifikaci (zdrojová IPv4 adresa, zdrojový port, cílová IPv4 adresa, cílový port, protokol). Vyhledávat se začíná od kořene uzlu. Z hlavičky uzlu se zjistí počet provedených řezů v jednotlivých dimenzích a podle hodnot z hlavičky paketu se vypočítá

index uzlu, se kterým se bude ve vyhledávání pokračovat. Používá-li aktuální uzel *equi-dense cuts*, tak se přepočítá index potomka. V poli potomků je potom uložena i hlavička potomka. Jestli aktuální uzel používá *equi-sized cuts*, tak se musí hlavička potomka zjistit zvlášť. S tímto principem se postupně prochází strom, až se dojde k listu stromu. Z množiny pravidel, které list obsahuje, se vyberou ty pravidla, která pokrývají příchozí paket. Tyto pravidla jsou vrácena jako výsledek prohledávání v jednom stromu.

Takto se postupně projdou všechny stromy a jejich výsledky se uloží do pole, které se seřadí podle priority. Výsledkem klasifikace je potom první položka v poli s výslednými pravidly.

Kapitola 6

Měření

Implementovaný algoritmus EffiCuts je optimalizací algoritmu HyperCuts a jeho předchůdce HiCuts. Proto je tato část práce zaměřena na srovnání těchto třech algoritmů. Nastavení těchto třech algoritmů je ukázáno v tabulce 6.1. Při měření se předpokládá zabraná paměť jednotlivých částí z tabulky 6.4. Měření probíhalo ve frameworku NetBench na vybraných čtyřech množinách filtrovacích pravidel o 100, 240, 455, 956 pravidlech. Výsledky relevantních vlastností jsou v tabulce 6.3.

	Space-factor	Bucket-size	Max-cuts
EffiCuts	8	16	8
HyperCuts	4	16	
HiCuts	2	16	

Tabulka 6.1: Nastavení algoritmů

Hlavička uzlu	2 byty
Velikost pravidla	13 bytů
Hranice prostoru	16 bytů
Ukazatel na pravidlo	4 byty
Ukazatel na uzel	4 byty

Tabulka 6.2: Paměťová náročnost jednotlivých částí.

6.1 Srovnání paměťových nároků

EfiiCuts přinesl tři optimalizace, které mají za cíl zmenšit velké paměťové nároky HyperCuts. První dvě optimalizace *Separable Trees* a *Selective Tree Merging* řeší překrývání malých a velkých pravidel v pravidlovém prostoru tak, že množinu pravidel rozdělí na podmnožiny, u kterých bude docházet k replikaci pravidel co nejméně. *Equi-dense cuts* je další z optimalizací, která řeší různou hustotu pravidel v pravidlovém prostoru, která vede k tvorbě neefektivních uzlů.

V tabulce 6.4 jsou porovnané celkové paměťové nároky na čtyřech množinách pravidel. Je zde vidět několikanásobná úspora paměti, která se zvyšuje s počtem filtrovacích pra-

fw2_05_m05_100 (100 pravidel)	EffiCuts	HyperCuts	HiCuts
Uložená pravidla/ukazatele	118	462	2864
Výška stromu	11	7	9
Počet uzlů	17	87	201
Celková velikost (bytů)	1 646	5 497	12 658
fw2_05_m05_250 (240 pravidel)	EffiCuts	HyperCuts	HiCuts
Uložená pravidla/ukazatele	316	2 540	34 064
Výška stromu	35	16	22
Počet uzlů	89	496	2 317
Celková velikost (bytů)	4 796	24 792	150 154
fw2_05_m05_500 (455 pravidel)	EffiCuts	HyperCuts	HiCuts
Uložená pravidla/ukazatele	485	5 155	50 064
Výška stromu	39	15	22
Počet uzlů	108	947	3 385
Celková velikost (bytů)	7 145	50 493	220 562
fw2_05_05 (956 pravidel)	EffiCuts	HyperCuts	HiCuts
Uložená pravidla/ukazatele	956	115 436	454 048
Výška stromu	11	7	11
Počet uzlů	287	14 991	32 645
Celková velikost (bytů)	15 175	721 981	2 012 058

Tabulka 6.3: Srovnání relevantních vlastností jednotlivých algoritmů.

videl. Na těchto čtyřech množinách pravidel potřeboval EffiCuts průměrně 16 krát menší paměť než HyperCuts a 51 krát menší paměť než HiCuts. Takto velká redukce celkové paměti je taká zapříčiněna množinou testovacích pravidel, která obsahuje velký počet velkých pravidel, které se často replikují. Poměr replikujících se pravidel se vypočítá jako počet uložených pravidel (EffiCuts) nebo ukazatelů na pravidla (HyperCuts) děleno počtem filtrovacích pravidel. Tento výpočet je proveden v tabulce 6.5, kde je vidět pouze nepatrná replikace pravidel u EffiCuts oproti ostatním algoritmům. Na obrázku 6.1 je vidět potřebný počet bytů pro uchování jednoho pravidla pro jednotlivé množiny pravidel (osa y je v logaritmickém měřítku). Je zde vidět, že počet bytů potřebných na uložení jednoho pravidla u EffiCuts zůstává téměř stejná.

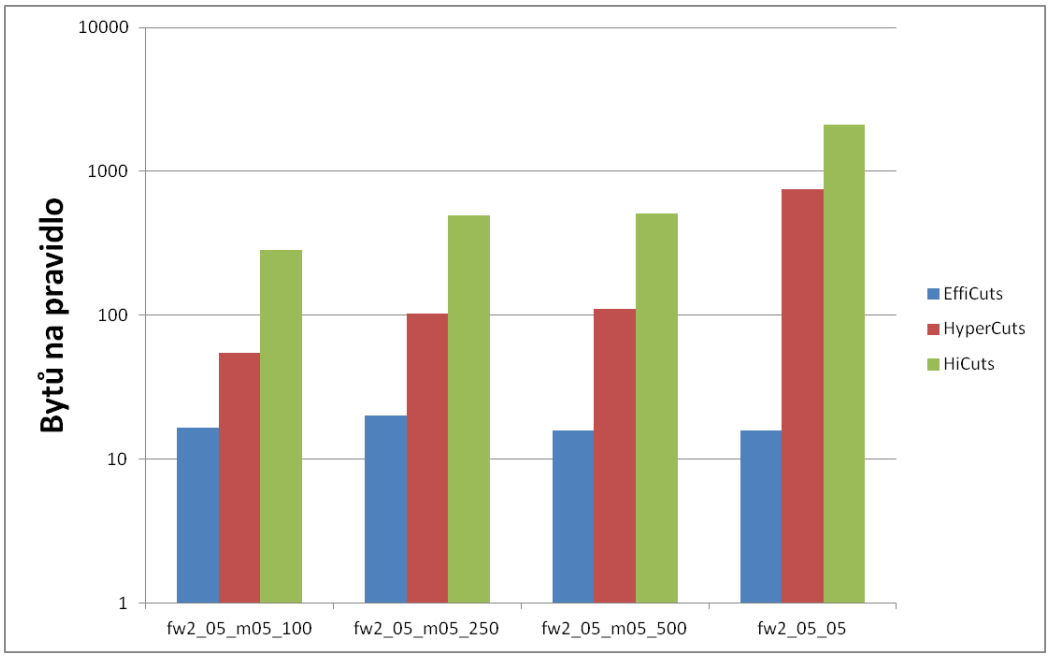
Počet pravidel	100	240	455	956
EffiCuts	1 646	4 796	7 145	15 175
HyperCuts	5 497	24 792	50 493	721 981
HiCuts	12 658	150 154	220 562	2 012 058
Redukce paměti oproti HyperCuts	3,34	5,17	7,07	47,56
Redukce paměti oproti HiCuts	7,69	31,3	30,87	132,59

Tabulka 6.4: Redukce paměti oproti HyperCuts a HiCuts.

EffiCuts výrazně snížil paměťovou náročnost. Má sice o něco větší celkovou výšku stromu, ale 97% uzlů používá *equi-dense cuts* a na to je aplikovaná čtvrtá optimalizace

Počet pravidel	EffiCuts	HyperCuts	HiCuts
100	1,18	4,62	28,64
240	1,32	10,58	141,93
455	1,07	11,33	110,03
956	1,06	120,749	474,95
Průměr	1,16	36,82	188,89

Tabulka 6.5: Rreplikace pravidel.



Obrázek 6.1: Potřebný počet bytů k uložení jednoho pravidla.

Node Co-location, která zredukuje potřebný počet přístupů do paměti ze dvou na jeden. Časová náročnost vyhledávání se téměř nezmění. V příloze jsou podrobné statistiky ze všech množin pravidel z knihovny NetBench.

Kapitola 7

Závěr

V této bakalářské práci jsem rozebral čtyři vysokoúrovňové přístupy ke klasifikaci paketů. Ke každému z těchto přístupů jsem nastudoval a popsal principy několika klasifikačních algoritmů a jejich hlavních vlastností.

Dále byl detailně popsán algoritmus pro klasifikaci paketů ve více dimenzích EfiiCuts, který vychází z algoritmu HyperCuts. Práce uvádí princip těchto algoritmů a zaměřuje se na optimalizace tohoto principu, které řeší příčiny velkých paměťových nároků při zachování téměř stejné propustnosti.

Výsledkem této práce je softwarová implementace algoritmu EffiCuts v jazyku Python. Pro implementaci byl využit framework NetBench sloužící k experimentování s algoritmy, které se zabývají problematikou zpracování paketů. Implementovaný algoritmus má sloužit k rozšíření knihovny již existujících klasifikačních algoritmů v tomto frameworku.

S implementovaným algoritmem EffiCuts bylo provedeno měření jeho podstatných vlastností na všech množinách pravidel z knihovny NetBench. Výsledné statistiky byly porovnány s algoritmy HyperCuts a HiCuts. Výsledky ukázaly, že EffiCuts potřebuje několikanásobně menší paměť. Hlavní příčinou malých paměťových nároků je téměř úplné odstranění replikujících se pravidel. Následkem toho se počet bytů potřebných k uložení jednoho pravidla téměř nemění s počtem filtrovacích pravidel.

Literatura

[1] *Michal Myška: Algoritmy pro klasifikaci paketů, bakalářská práce, Brno, FIT VUT v Brně, 2012.*

[2] *Viktor Puš: Klasifikace paketů s využitím technologie FPGA, diplomová práce, Brno, FIT VUT v Brně, 2008.*

[3] Balajee Vamanan, Gwendolyn Voskuilen, T. N. Vijaykumar: *EffiCuts: Optimizing Packet Classification for Memory and Throughput.* SIGCOMM, 2010.

[4] Lakshman, T. V. and Stiliadis, D: *High-speed policy-based packet forwarding using efficient multi-dimensional range matchingl.* ACM Sigcomm, 1998.

[5] Matoušek, P.: *Klasifikace paketů a filtrování dat.* FIT VUT Brno, 2012.

[6] MONTOYE, R. K: *Apparatus for storing "Don't Care" in a content addressable memory cell.* HaL Computer Systems, Inc., 1994.

[7] Pankaj Gupta and Nick McKeown: *Packet Classification on Multiple Fields.* ACM SIGCOMM, 1999.

[8] Pankaj Gupta and Nick McKeown: *Packet Classification using Hierarchical Intelligent Cuttings.* IEEE Symposium on High Performance Interconnects (HotI), 1999.

[9] Song, H.: Evaluation of Packet Classification Algorithms. `http://www.arl.wustl.edu/~hs1/`.

[10] Srinivasan, V., Suri, S., Varghese, G: *Packet classification using tuple space searchl.* ACM Sigcomm, 1999, 135–146 s.

[11] Srinivasan, V., Suri, S., Varghese, G., Wadvoge: *Fast and scalable layer four switchingl.* ACM Sigcomm, 1998.

[12] Sumeet Singh, Florin Baboescu, George Varghese, Jia Wang: *Packet Classification using Multidimensional Cutting.* ACM SIGCOMM, 2003.

[13] Taylor, D. E.: *Survey and Taxonomy of Packet Classification Techniques.* ACM Computing Surveys, 37(3), 2005, 238–275 s.

[14] Viktor Puš, Vlastimil Košař, Jiří Tobola: *Netbench–Design Document.* Faculty of Information Technology, Brno University of Technology. Brno 2012.

Příloha A

Obsah CD

K této bakalářské práci je přidáno CD, které obsajuje zdrojový kód algoritmu EffiCuts, soubory potřebné k vytvoření technické zprávy v LaTeXu, pdf s výsledky algoritmu EffiCuts ze všech množin pravidel z frameworku NetBench a elektronickou verzi technické zprávy v pdf.